

Data Wrangling with Hadoop and MongoDB

Case Study Analysis of Nike Audience using Tweet Analysis

**Submitted By:**

Swati Singh

Contents

[Executive Summary 2](#_Toc5302932)

[Sandbox Installation 3](#_Toc5302933)

[Introduction 3](#_Toc5302934)

[Prerequisites 3](#_Toc5302935)

[Installation 3](#_Toc5302936)

[Setup Virtual Machine Configuration 3](#_Toc5302937)

[Deploy HDP Sandbox 3](#_Toc5302938)

[Deploy HDF Sandbox 4](#_Toc5302939)

[Installation Checks 5](#_Toc5302940)

[Map Sandbox IP to hostname 5](#_Toc5302941)

[Start the sandboxes 5](#_Toc5302942)

[Important Instructions 7](#_Toc5302943)

[Creating Twitter API Application 8](#_Toc5302944)

[Setup Development Environment 9](#_Toc5302945)

[Prerequisites 9](#_Toc5302946)

[Setup HDF Environment 10](#_Toc5302947)

[Nifi 10](#_Toc5302948)

[Kafka 11](#_Toc5302949)

[Setup HDP Environment 12](#_Toc5302950)

[Kafka 12](#_Toc5302951)

[HDFS 12](#_Toc5302952)

[Spark 13](#_Toc5302953)

[Create Nifi Data Flow 14](#_Toc5302954)

[Deep Dive into DataFlow 1 15](#_Toc5302955)

[Deep Dive into DataFlow 2 17](#_Toc5302956)

[Start Process Group GetNikeTweets 18](#_Toc5302957)

[Building Sentiment Analysis Model 19](#_Toc5302958)

[Prerequisites 20](#_Toc5302959)

[Load data into Spark 20](#_Toc5302960)

[Gradient Boosting Classification Model 22](#_Toc5302961)

[Spark Streaming Application to Deploy Model 26](#_Toc5302962)

[Data Analysis in MongoDB 29](#_Toc5302963)

[Resources 36](#_Toc5302964)

# Executive Summary

What differentiates and delivers values over saying “you’ve got more leads and you’ve got more traffic” is actually explaining why. Tell the story and provide insight. Lee Odden

Big data analytics and powerful computing services are providing intangible values to business strategy development and enhance customer experience. Big Data refers to complex and large data sets that have to be processed and analyzed to uncover valuable information that can benefit businesses and organizations. There are various forms of big data: Structured, unstructured and semi structured data.

This project is focused on leveraging multiple big data technologies provided by Hortonworks Data Flow and Hortonworks Data Platform to build a real-time Sentiment Analysis Application. Sentiment Analysis is the analysis of body of text to identify and extract opinion bearing words or sentiments expressed by the text. It is one of the most popular applications of Natural Language Processing(NLP).

The first step to big data analytics is gathering the data itself. This is known as “data mining.” In this project, we will be exploring how to use data mining technique to gather twitter data and stream into ‘Kafka topic’ using NiFi. Next, we will build a spark/scala machine learning algorithm that assigns polarity values from 0 and 1 to the tweets. Values equal to 1 indicate more positivity, while values equal to 0 indicate more negativity.

Once the model is built, we will use Spark Structured Streaming to load the model from HDFS, pull in tweets from Kafka topic “tweets”, add a sentiment score to the tweet, then stream the data to Kafka topic “tweetsSentiment”. We will stream data from ‘tweetsSentiment’ to MongoDB using another NiFi dataflow. With MongoDB Aggregation framework, we will perform visualization to gather insights from the tweet sentiment data.

Twitter analytics provides wealth of information in forming strategies to drive design marketing campaigns, plan business developments, enrich user engagement and understand audience perception. For this project, analysis on Nike brand perception is performed to answer few questions:

1. What is the sentiment around the brand on twitter platform as a trend
2. Who are the most popular influencers for the Nike Brand
3. What are the most popular hashtags?
4. What are the number of followers over time?
5. Which are the most popular/unpopular tweets?

These questions can help understand what actions of the company are affecting user sentiment on a daily basis.

# Sandbox Installation

## Introduction

In this installation guide, we have described steps to setup HDP and HDF sandbox using Docker. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. With Docker, Apache Hadoop has packaged its sandbox with all the parts it needs, such as libraries and dependencies and ships it all out as one image package.

HDP (Hortonworks Data Platform) is a suite of tools to process data at rest. It stores, process and analyzes large volumes of data. The platform is designed to deal with data from many sources and formats.

HDF(Hortonworks Data Flow) contains the components which ingests, curates and analyzes real time flow /streaming data for key insights and immediate actionable intelligence.

## Prerequisites

* Installed [Docker](https://www.docker.com/products/docker-desktop)
* Minimum **10GB RAM** dedicated to docker-machine
* Installed Bash Shell. [Git-Bash](https://git-scm.com/downloads) is recommended
* Script Editor – [Atom](https://atom.io/) is recommended
* Download and Extract latest scripts [Hortonworks Data Platform](https://www.cloudera.com/downloads/hortonworks-sandbox/hdp.html) in **Downloads** folder
* Downloaded and extracted [Hortonworks Data Flow](https://www.cloudera.com/downloads/hortonworks-sandbox/hdf.html) in **Downloads** folder

## Installation

### Setup Virtual Machine Configuration

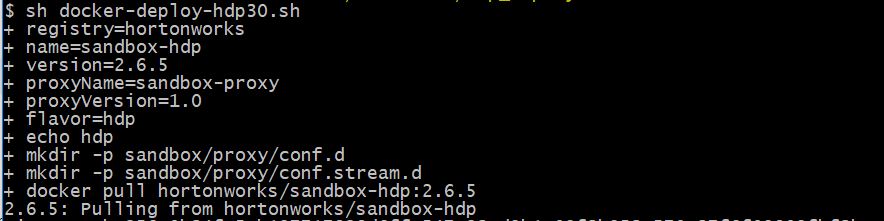
Open the docker application and execute the following commands

1. docker-machine rm default
2. docker-machine create -d virtualbox --virtualbox-disk-size "100000" --virtualbox-memory "10240" --virtualbox-cpu-count "4" default
3. #Optional 3&4
4. docker-machine ssh default "sudo mkdir /sys/fs/cgroup/systemd"
5. docker-machine ssh default "sudo mount -t cgroup -o none,name=systemd cgroup /sys/fs/cgroup/systemd"

### Deploy HDP Sandbox

|  |  |  |
| --- | --- | --- |
| 1. Before deploying the script, ensure the HDP version is set as 2.6.5 | |  |
| **Connected Data Architecture** is currently supported by HDP 2.6.5 and HDF version 3.1.1. Hortonworks Connected Data Architecture (CDA) is composed of both Hortonworks DataFlow (HDF) and Hortonworks DataPlatform (HDP) sandboxes and allows us to play with both data-in-motion and data-at-rest frameworks simultaneously.   1. Run Git Bash in the decompressed HDP scripts folder and execute the following command   sh docker-deploy-hdpXX.sh | alt text | |

Th script execution should look something like this



### Deploy HDF Sandbox

Like HDP Deployment, execute decompressed HDF deployment script using Git Bash

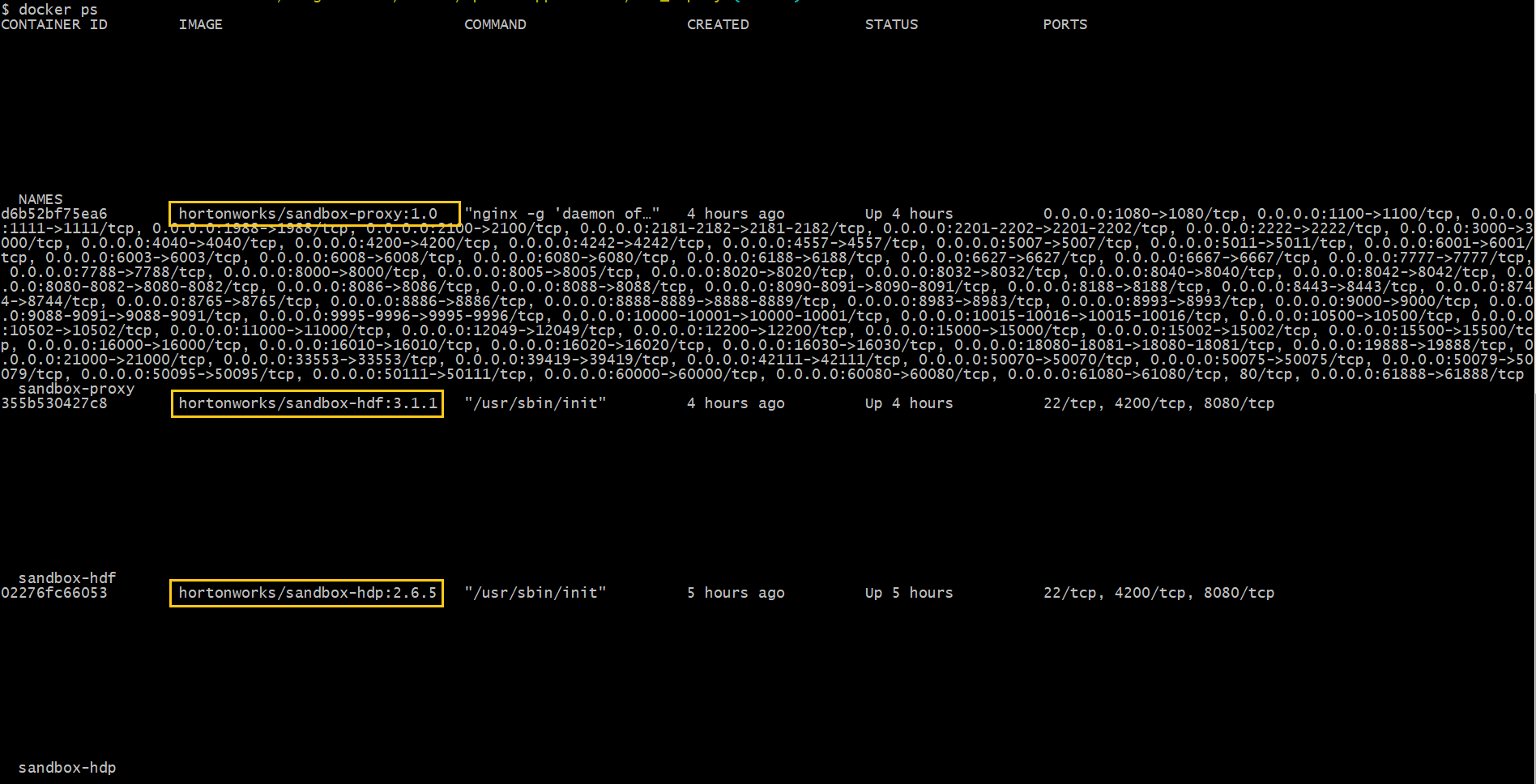
sh docker-deploy-hdfXXX.sh

|  |  |
| --- | --- |
| The script execution should look something like this.**->**  After the installation of HDP and HDF , enable CDA by executing the enable-native-cda script in this folder.  sh enable-native-cda.sh | alt text |

### Installation Checks

Now that both HDP and HDF sandbox have been deployed, we can see the containers running in docker as follows:

$ docker ps



The first is the NGINX proxy container followed by HDF and HDP containers. NGINX Proxy is a reverse proxy server that resides behind a firewall and directs incoming requests to multiple backend servers. In our case, it is HDP and HDF containers.

### Map Sandbox IP to hostname

We will now map the IP address of the docker machine with that of the hostname. Remember, the Nginx reverse proxy server directs the request to the appropriate container. Here, based on the hostname, appropriate sandbox container will be accessed.

1. #IP Address of the docker machine
2. $ docker-machine ip
3. XXX.XXX.XX.XXX
4. Run Notepad as administrator.
5. Open hosts file located in: c:\Windows\System32\drivers\etc\hosts
6. Add {IP-Address} localhost sandbox-hdp.hortonworks.com sandbox-hdf.hortonworks.com
7. Save the file
8. IMPORTANT: Replace {IP-Address} with Sandbox IP Address

### Start the sandboxes

For first time that we login, we need to reset the hdp and ambari admin password

1. ssh root@sandbox-hdp.hortonworks.com -p 2222
2. <Enter password : Hadoop>
3. <Prompt to enter new password>
4. <Re-enter new passoword>
5. ambari-admin-password-reset
6. <Enter new hdp ambari password>

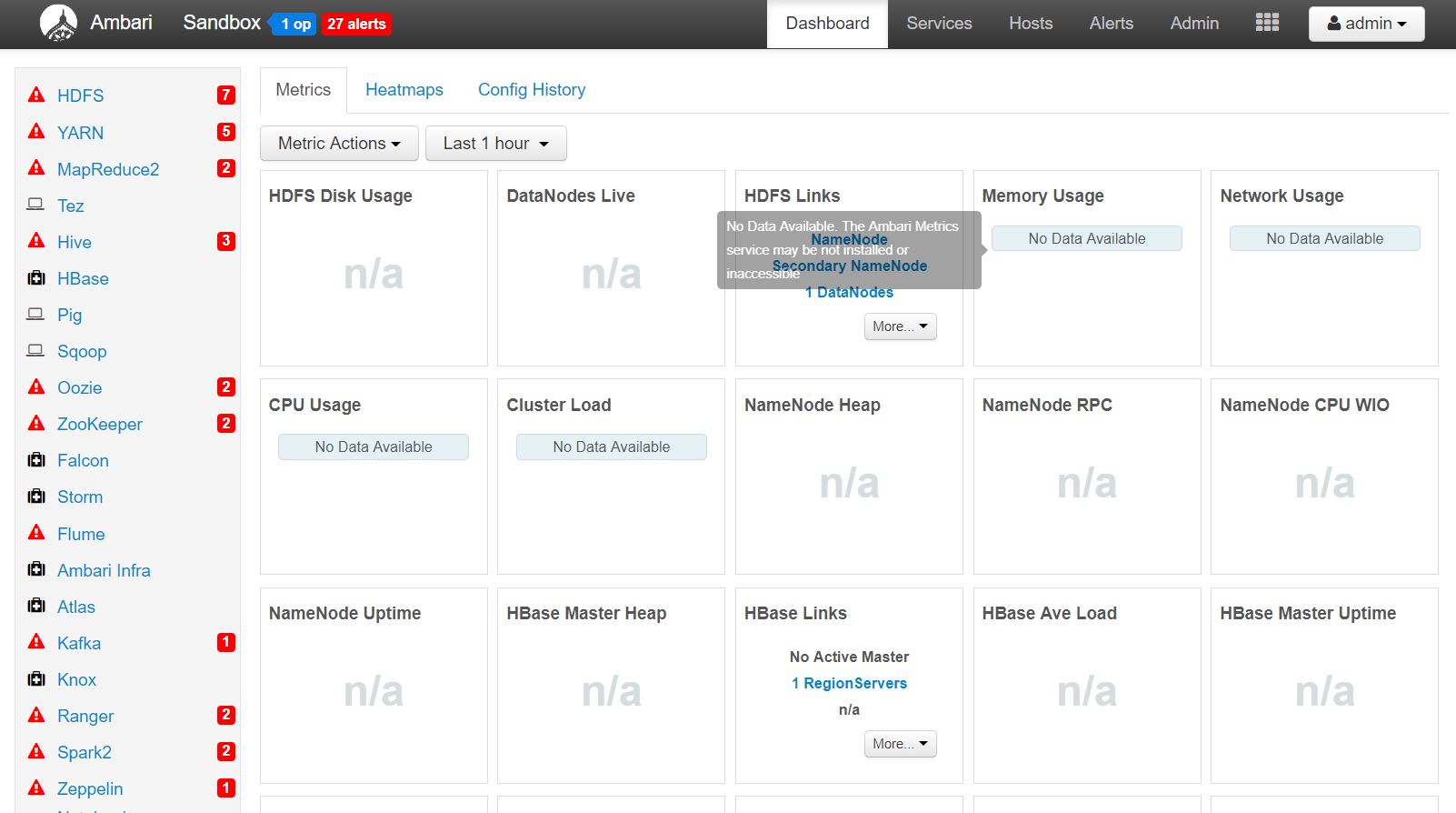
 Wait for the ambari server to restart

Follow similar steps to reset HDF Ambari password. Replace ssh root@sandbox-hdp.hortonworks.com -p 2222 with 1. ssh root@sandbox-hdf.hortonworks.com -p 2222

For login credentials for other profile, follow reference guide [here](https://hortonworks.com/tutorial/learning-the-ropes-of-the-hortonworks-sandbox/#login-credentials)

#### HDP Sandbox

<http://sandbox-hdp.hortonworks.com:8080/>



#### HDF Sandbox

<http://sandbox-hdf.hortonworks.com:8080/>



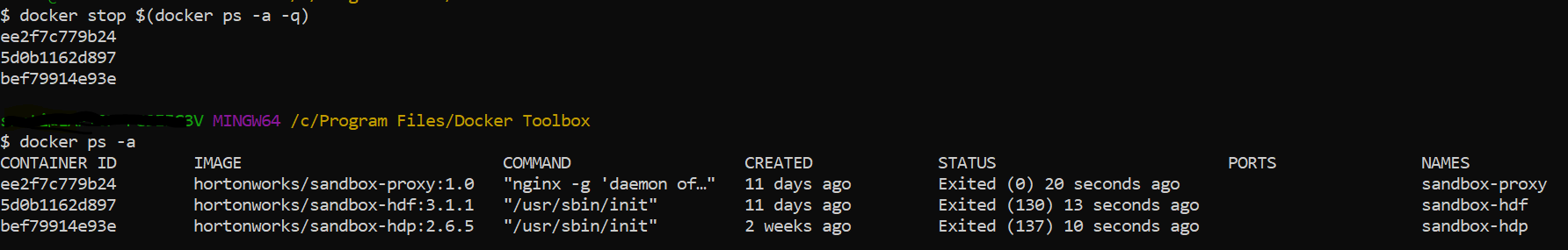
**Note**: At this moment, we do not care what services are running and which ones have stopped. We will restart necessary services and stop the rest to optimize memory utilization and improve processing performance.

### Important Instructions

When not running this project, stop all docker containers. The system allocates memory for the execution of the virtual machine which is running these containers. If the containers remain in execution stage, the system’s resources will remain busy and may hinder the performance of the entire system.

**Stop all docker containers**

**docker stop $(docker ps -a -q)**



**Start all docker containers**

**docker start $(docker ps -a -q)**



**Check all container statuses**

**docker ps -a**



**Start individual container**

Docker start <container-name:NAMES)



# Creating Twitter API Application

Twitter's developer portal includes numerous API endpoints and tools that helps us build an app on Twitter. For this project, we will use the Twitter Search API to fetch tweets matching a specified query.

Visit the developer portal with an approved developer account to create a Twitter App and generate our authentication tokens. These include 'consumer' tokens and secrets for app authentication, and 'access' tokens and secrets for user/account authentication.

Click on 'Create an App'.

Fill in required fields:

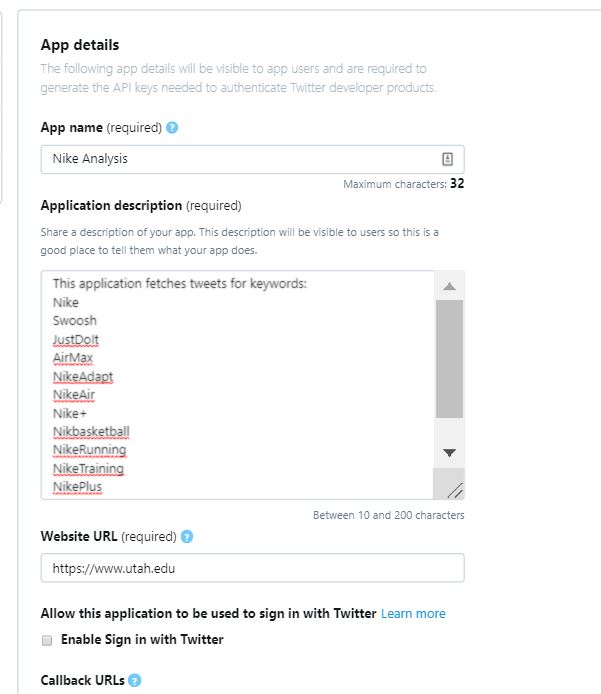
App Name

Application Description

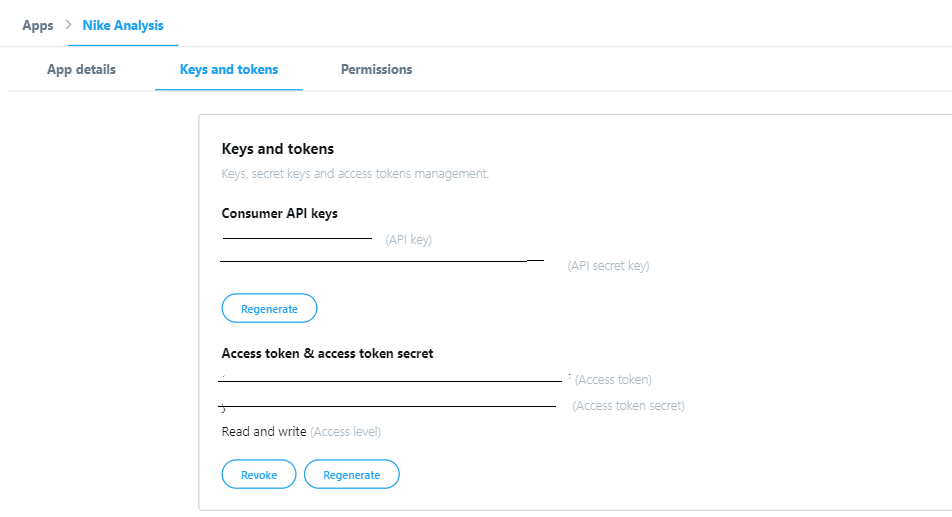
Website URL

Tell us how this app will be used Click on 'Create'

Following is an example of required fields to be filled:



We can now access our authentication tokens



We will use this to fetch and store Twitter data in kafka topic at a later stage.

# Setup Development Environment

## Prerequisites

1. Map hostname with Sandbox IP
2. Setup Ambari Admin password for HDP and HDF
3. Acquire Twitter authorization tokens

We will need to setup various components of the HDP and HDF suite to collect, process and analyze our data. Some of the components we will be touching are:

* Nifi
* Kafka
* HDFS
* Spark

Before we start setting up the development environment, ensure that all required services are started in Ambari

|  |  |
| --- | --- |
| Figure 1 HDF Services | Figure 2 HDP Services |

## Setup HDF Environment

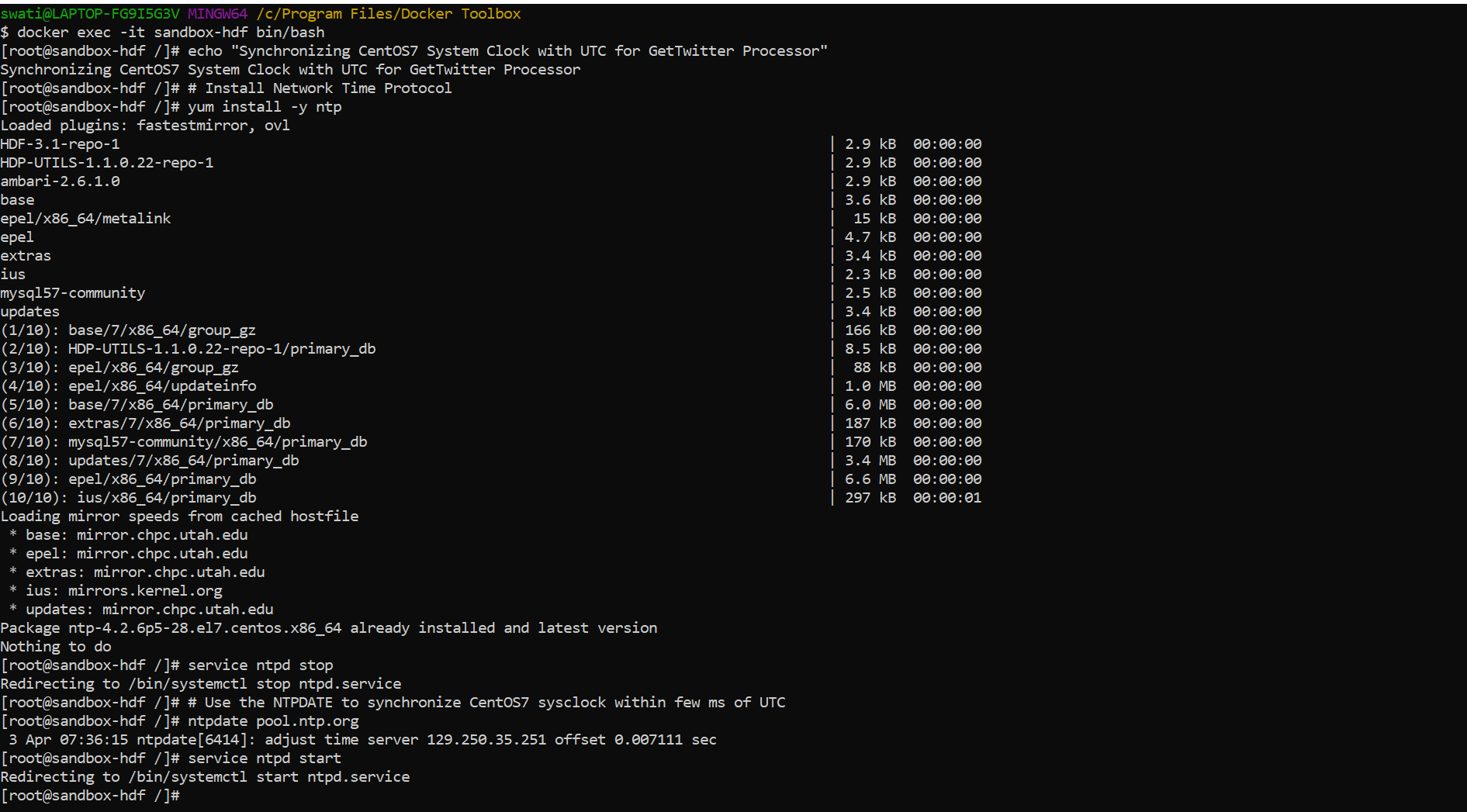
### Nifi

Nifi is a platform to automate flow of data between different systems. It provides real-time streaming capabilities that allows a user to send, receive, route, transform, and sort data, as needed, in an automated and configurable way.

We will build two Nifi dataflows. The first dataflow will connect to the Twitter API and pull desired attributes in our kafka topic ‘tweets’.

The Twitter API stores and returns tweets in GMT. It is important to synchronize the system clock with GMT to avoid running into authorization errors while connecting to the Twitter API Feed through GetTwitter processor. Another reason to synchronize the clocks is that since we will collect the tweets in batches at an interval of few seconds, we may lose some tweets that may have appeared since the last updated time.

1. $ docker exec -it sandbox-hdf bin/bash
2. [root@sandbox-hdf /]# echo "Synchronizing CentOS7 System Clock with UTC for GetTwitter Processor"
3. [root@sandbox-hdf /]# # Install Network Time Protocol
4. [root@sandbox-hdf /]# yum install -y ntp
5. [root@sandbox-hdf /]# service ntpd stop
6. [root@sandbox-hdf /]# # Use the NTPDATE to synchronize CentOS7 sysclock within few ms of UTC
7. [root@sandbox-hdf /]# ntpdate pool.ntp.org
8. [root@sandbox-hdf /]# service ntpd start



The second part of the code cleans up the NiFi flow that is already prebuilt into HDF Sandbox by backing up the flow and removing it.

1. mv /var/lib/nifi/conf/flow.xml.gz /var/lib/nifi/conf/flow.xml.gz.bak

### Kafka

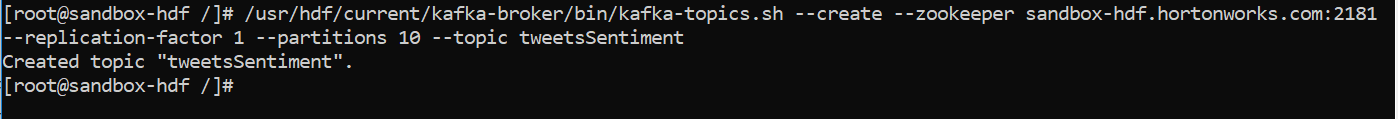
In order to stay competitive, businesses today rely increasingly on real-time data analysis allowing them to gain faster insights and quicker response times. Apache takes information – which can be read from a huge number of data sources – and organizes it into “topics”. This is achieved using a function known as a Producer, which is an interface between applications (twitter API) and the topics – Kafka’s own database of ordered, segmented data, known as the Kafka Topic Log.

Often this data stream will be used to fill data lakes such as Hadoop’s distributed databases or to feed real-time processing pipelines like Spark or Storm.

Another interface – known as the Consumer – enables topic logs to be read, and the information stored in them passed onto other applications which might need it, in our case, the sentiment analysis application.

We will need to create a Kafka topic ‘tweetsSentiment’ on HDF for Spark to stream data into from HDP.

1. /usr/hdf/current/kafka-broker/bin/kafka-topics.sh --create --zookeeper sandbox-hdf.hortonworks.com:2181 --replication-factor 1 --partitions 10 --topic tweetsSentiment
2. Exit

**Note**: Restart NiFi and Kafka for the changes to take effect.

## Setup HDP Environment

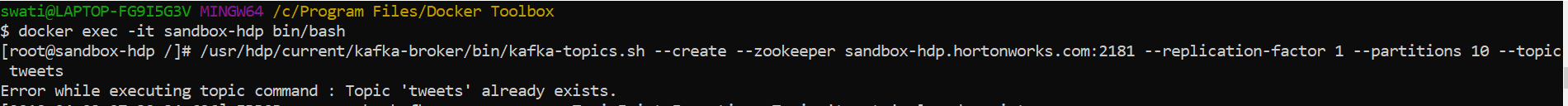
To enter HDP bash, execute the following command

1. docker exec -it sandbox-hdp bin/bash

### Kafka

Create a Kafka topic on HDP for NiFi to publish tweets to the Kafka queue.

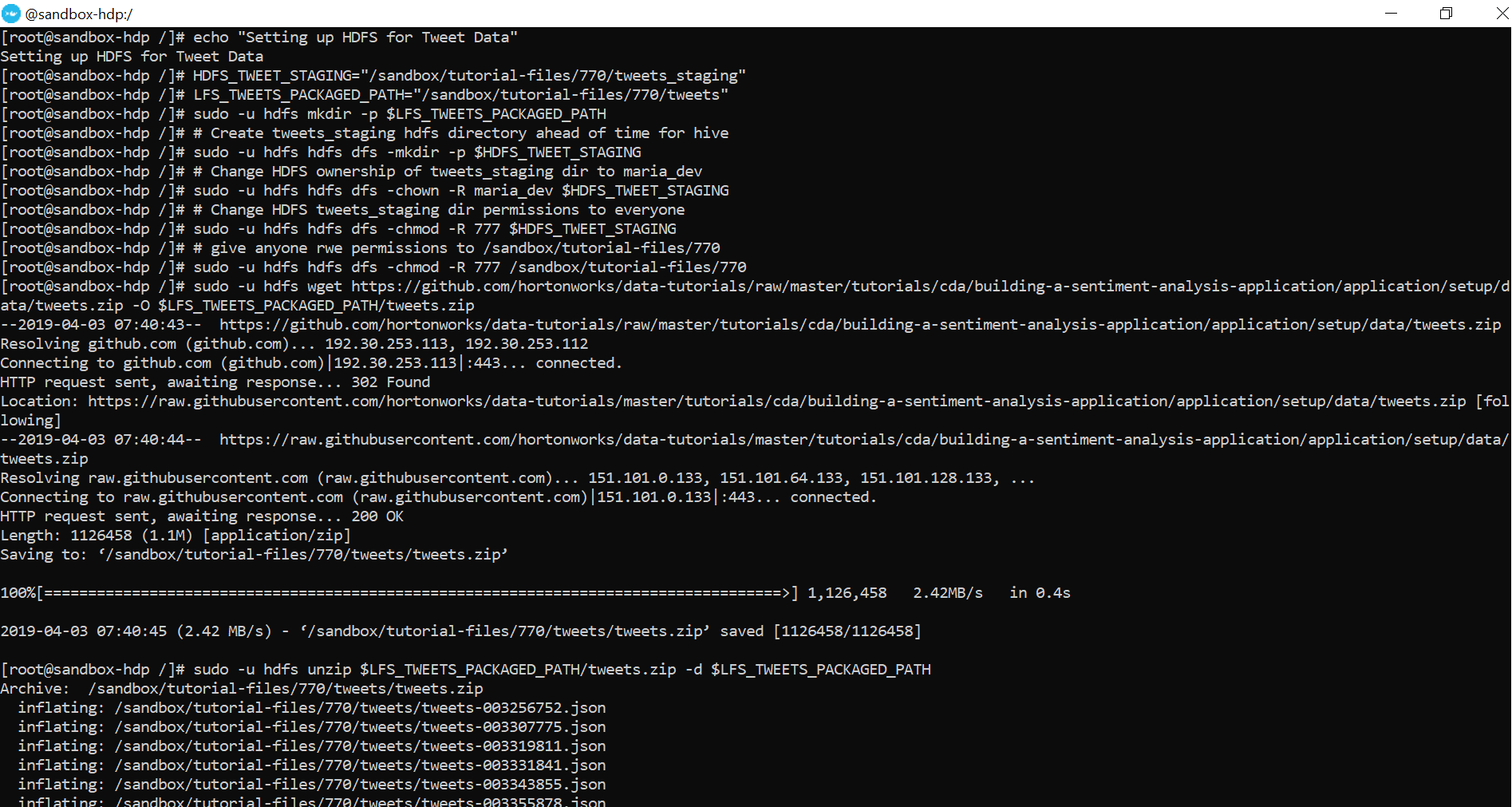
1. /usr/hdp/current/kafka-broker/bin/kafka-topics.sh --create --zookeeper sandbox-hdp.hortonworks.com:2181 --replication-factor 1 --partitions 10 --topic tweets



### HDFS

The Hadoop Distributed File System (HDFS) is the primary data storage system used by Hadoop applications. We will now create tweets folder that will hold a zipped file. This data will be copied over to HDFS where it will be later loaded by Spark to refine the historical data for creating a machine learning model.

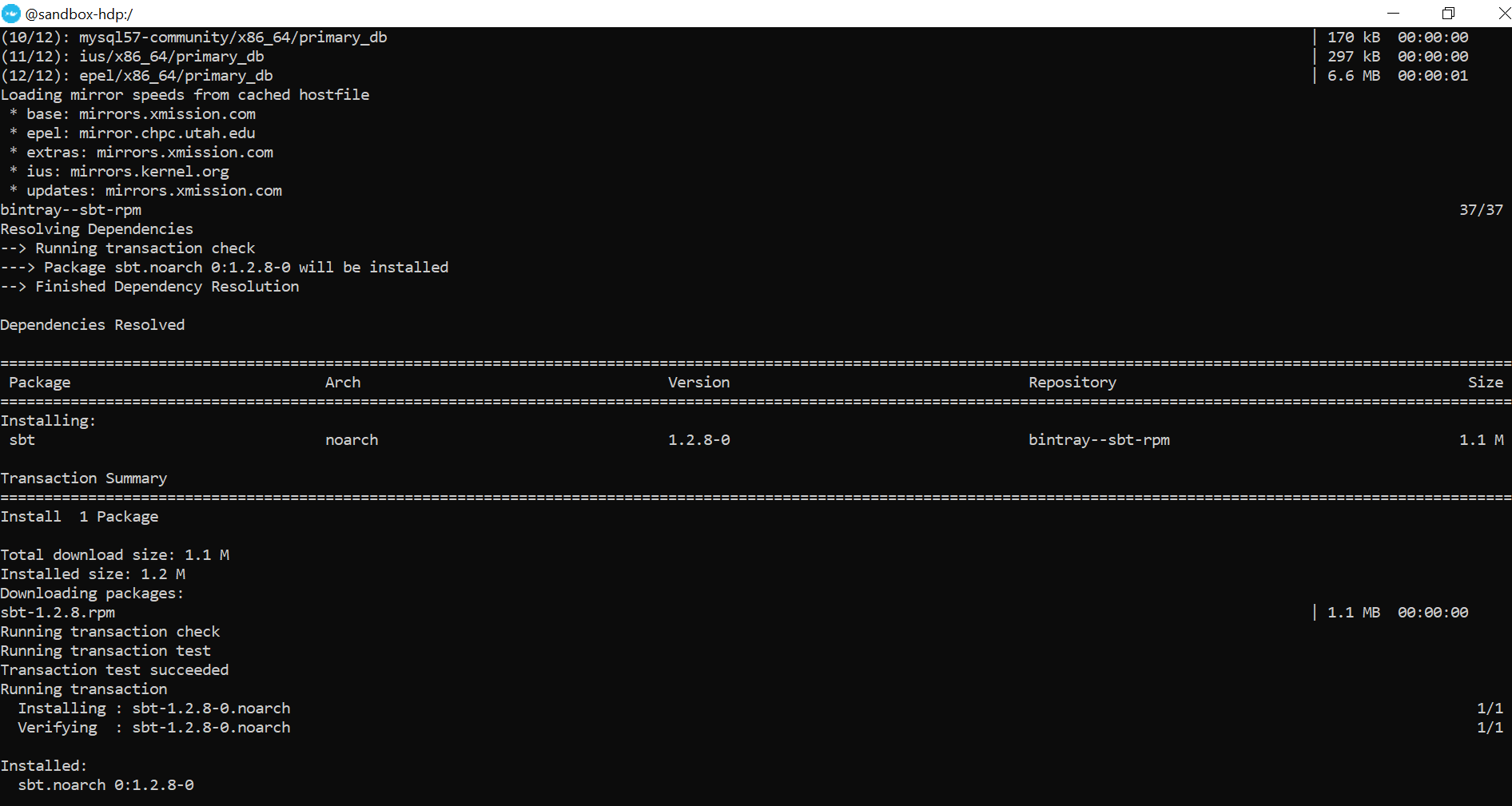
1. echo "Setting up HDFS for Tweet Data"
2. HDFS\_TWEET\_STAGING="/sandbox/tutorial-files/770/tweets\_staging"
3. LFS\_TWEETS\_PACKAGED\_PATH="/sandbox/tutorial-files/770/tweets"
4. sudo -u hdfs mkdir -p $LFS\_TWEETS\_PACKAGED\_PATH
5. # Create tweets\_staging hdfs directory ahead of time for hive
6. sudo -u hdfs hdfs dfs -mkdir -p $HDFS\_TWEET\_STAGING
7. # Change HDFS ownership of tweets\_staging dir to maria\_dev
8. sudo -u hdfs hdfs dfs -chown -R maria\_dev $HDFS\_TWEET\_STAGING
9. # Change HDFS tweets\_staging dir permissions to everyone
10. sudo -u hdfs hdfs dfs -chmod -R 777 $HDFS\_TWEET\_STAGING
11. # give anyone rwe permissions to /sandbox/tutorial-files/770
12. sudo -u hdfs hdfs dfs -chmod -R 777 /sandbox/tutorial-files/770
13. sudo -u hdfs wget https://github.com/hortonworks/data-tutorials/raw/master/tutorials/cda/building-a-sentiment-analysis-application/application/setup/data/tweets.zip -O $LFS\_TWEETS\_PACKAGED\_PATH/tweets.zip
14. sudo -u hdfs unzip $LFS\_TWEETS\_PACKAGED\_PATH/tweets.zip -d $LFS\_TWEETS\_PACKAGED\_PATH
15. sudo -u hdfs rm -rf $LFS\_TWEETS\_PACKAGED\_PATH/tweets.zip
16. # Remove existing (if any) copy of data from HDFS. You could do this with Ambari file view.
17. sudo -u hdfs hdfs dfs -rm -r -f $HDFS\_TWEET\_STAGING/\* -skipTrash
18. # Move downloaded JSON file from local storage to HDFS
19. sudo -u hdfs hdfs dfs -put $LFS\_TWEETS\_PACKAGED\_PATH/\* $HDFS\_TWEET\_STAGING



### Spark

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. This Spark Structured Streaming application will pull in JSON messages from Kafka topic “tweets”, adds a sentiment field to the JSON based on the sentiment model loaded in from HDFS and sends the enriched data back to Kafka topic “tweetsSentiment”. For Spark Structured Streaming, we will need to leverage SBT package manager. SBT compiles and runs a Scala project, and package the project as a JAR file. The commands below install SBT.

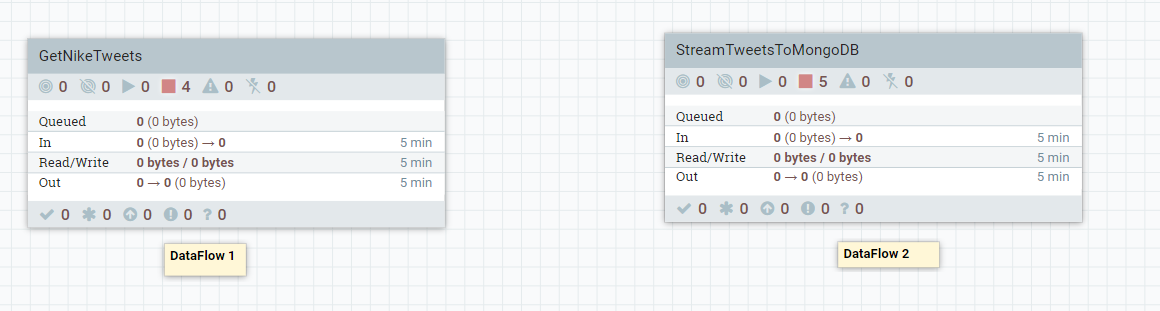
1. curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
2. yum install -y sbt

**Note**: Restart Kafka, HDFS, and Spark for the changes to take effect.

# Create Nifi Data Flow

We will build two dataflows to perform two separate functions

**DataFlow1**: This NiFi dataflow will ingest data from Twitter using the twitter credentials, pull key attributes that will help with our analysis and store the data in Kafka topic called tweets



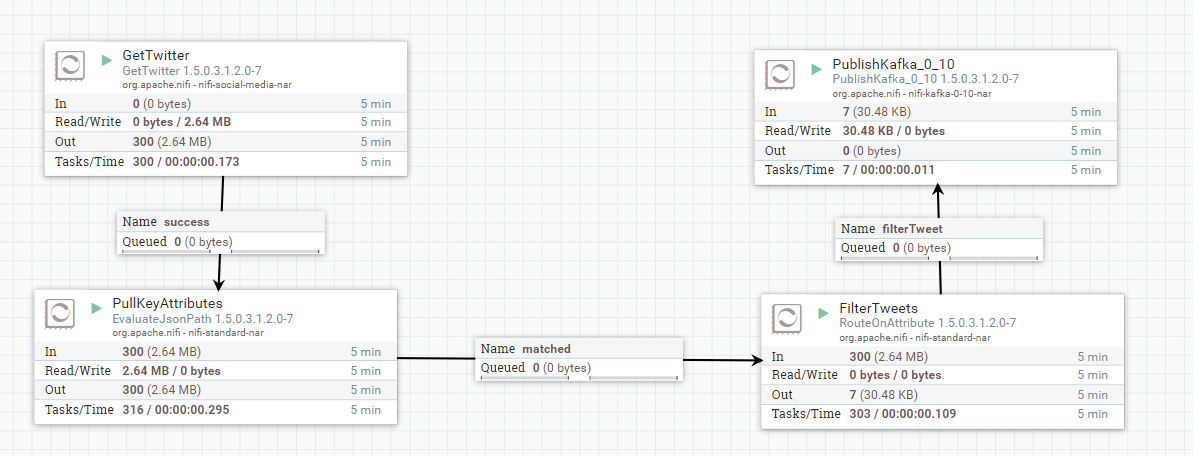
**DataFlow2**: The second NiFi flow in another process group will consume data from Kafka topic ‘tweetsSentiment’, which has a trained sentiment model built with an external service SparkML and send the data to be stored into MongoDB.

It is recommended to import the [NiFi flow](https://github.com/swatisingh0107/NikeRealTimeDataAnalysis/blob/master/NifiFlowFile/NikeNifiFlow.xml) from your local computer to the NiFi Interface.

1. Open the Nifi Interface: <http://sandbox-hdf.hortonworks.com:9090/nifi/>
2. Right Click to select ‘Upload Template’
3. Click on the search icon and select the XML file
4. Click on the Upload Button

## Deep Dive into DataFlow 1

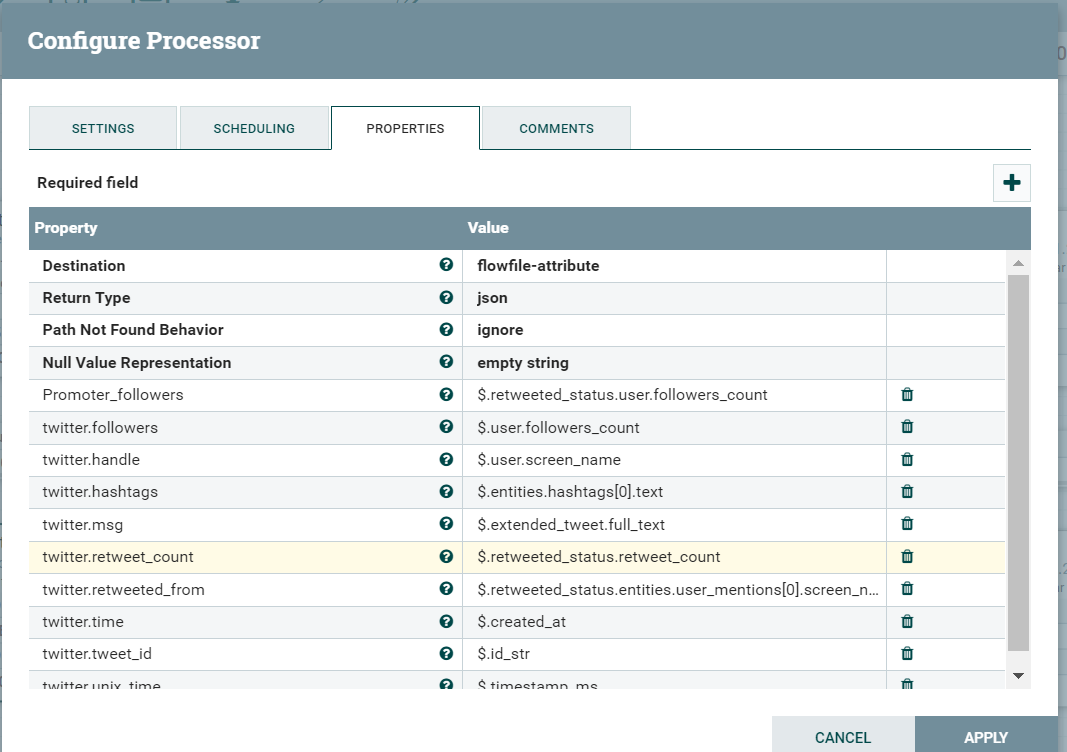
DataFlow1 Processor Group consists of four processors.



1. **GetTwitter**: In NiFi we have the in-built GetTwitter processor which pulls tweets through twitter streaming API. Double click on the processor to populate our twitter credentials and the keywords to filter tweets on.



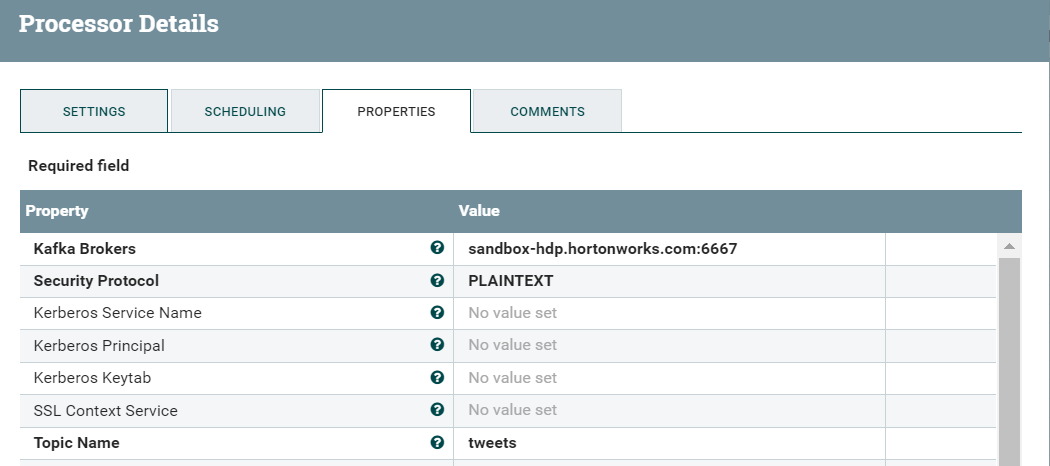
1. **EvaluateJSONPath**: EvaluateJsonPath is used to extract Json fields as attribute or content. Setting the Destination to flowfile-attribute ensures that each JSON Path will be extracted to the named attribute value. The twitter JSON values are the name of the keys of Tweet JSON data. More information about the Twitter JSON format can be found [here](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json).



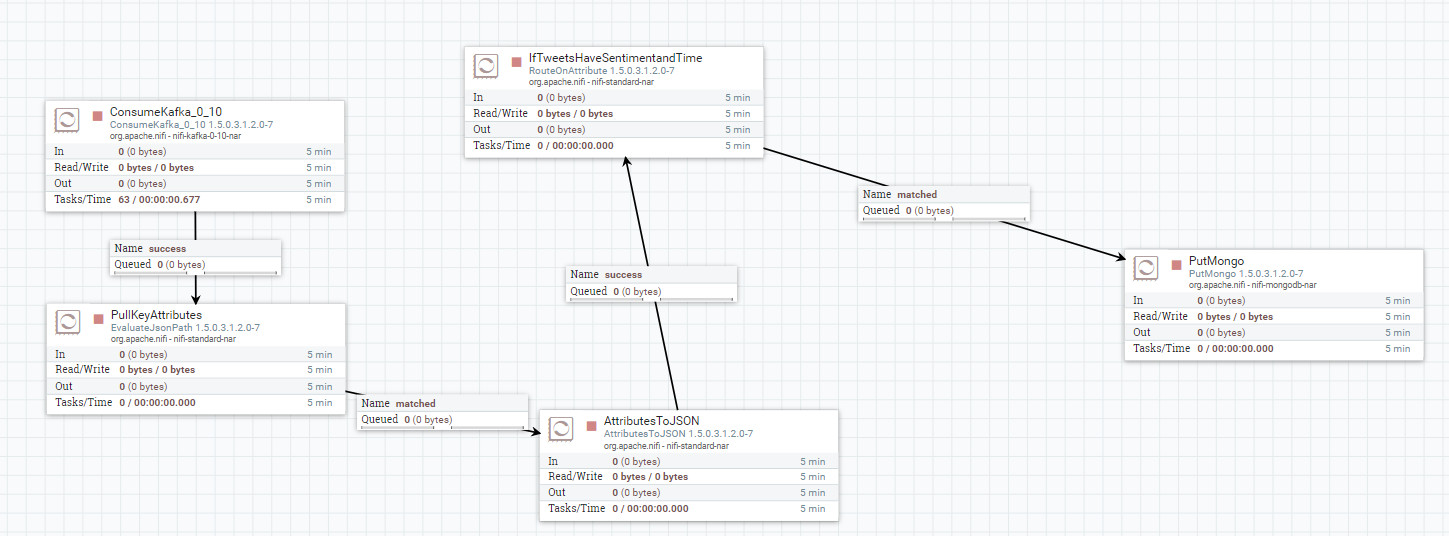
1. **RouteonAttribute:** One of the most powerful features of NiFi is the ability to route FlowFiles based on their Attributes. The primary mechanism for doing this is the RouteOnAttribute Processor. Any number of user-defined properties can be added by clicking the "+" button in the top-right corner of the Properties tab in the Processor's Configure dialog. The most common strategy is the "Route to Property name" strategy. With this strategy selected, the Processor will expose a Relationship for each property configured. If the FlowFile's Attributes satisfy the given expression, a copy of the FlowFile will be routed to the corresponding Relationship. For example, in our case, FlowFile will be routed if the tweet’s message/text is not empty. This ensure that we do not have redundant data.



**PublishKafka**: Finally, we extract the metadata, key and value. Publish kafka key and value using PublishKafka processor. Note, that kafka topic name ‘tweets’ should be updated in PublishKafka processor.

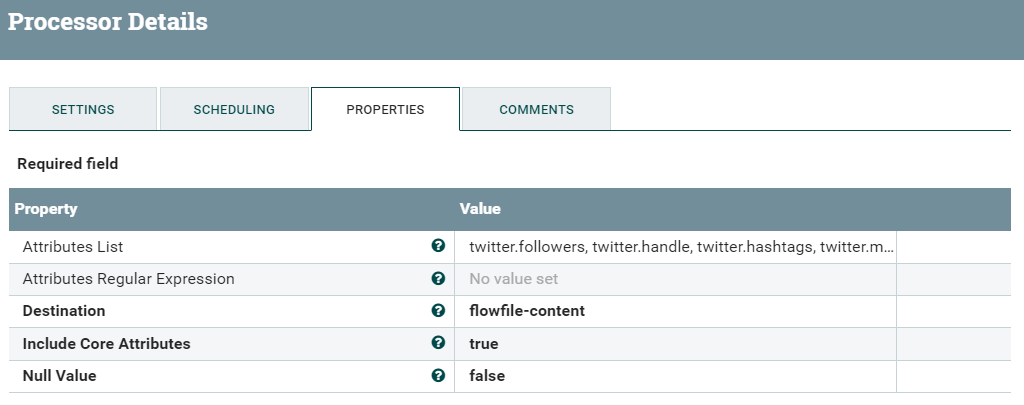


## Deep Dive into DataFlow 2



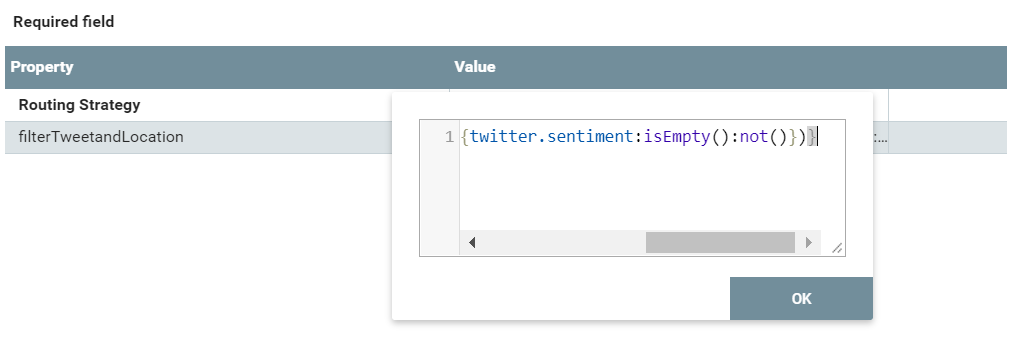
**ConsumeKafka**: This processor consumes messages from kafka topic ‘tweetsSentiment’ and generates flowfile for each message. The FlowFile moves to the EvaluateJSONPath which is a copy of the ‘PullKeyAttributes’ processor from DataFlow1.

**AttributesToJSON**: This processor generates a JSON representation of the input FlowFile Attributes. The resulting JSON will be written to a FlowFile as content.

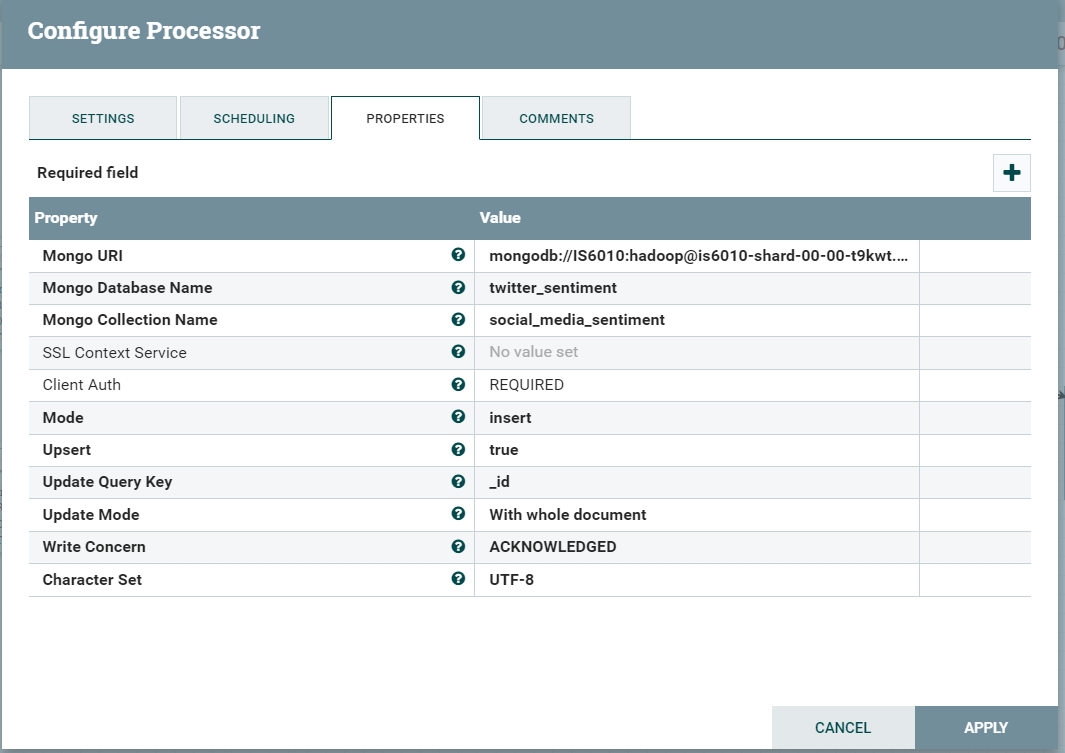


**RouteOnAttribute**: FlowFile will be routed if the tweet’s message has a timestamp and a sentiment score attached to it. This ensures that we have quality data to perform sentiment analysis.

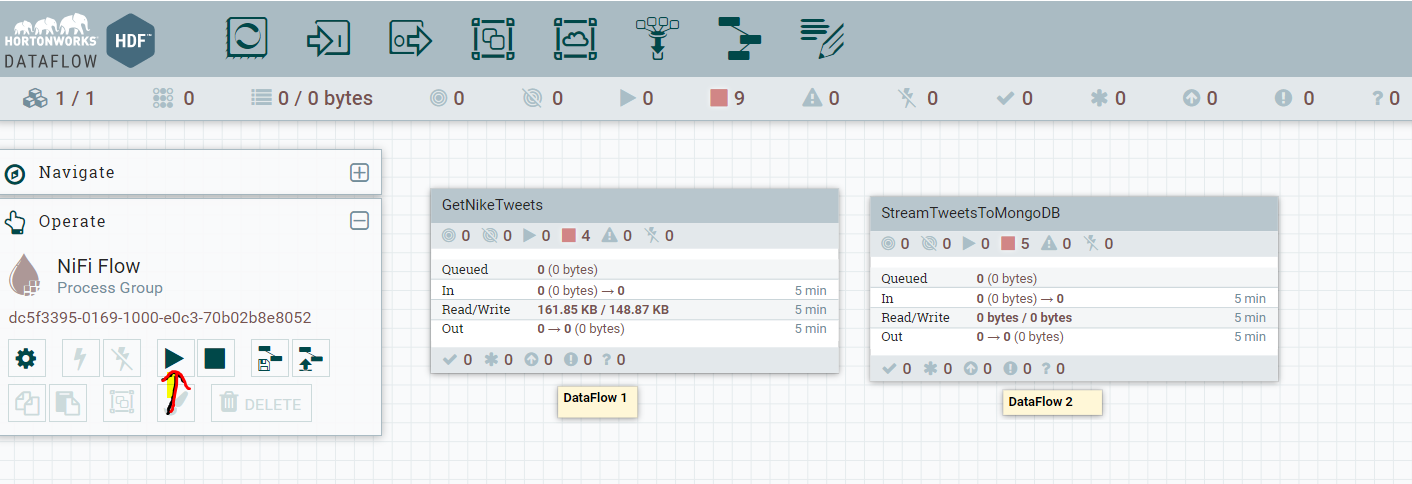
${twitter.unixtime:isEmpty():not():and(${twitter.sentiment:isEmpty():not()})}



**PutMongo**: Once verified that the message has a sentiment score attached to it, we will now write each JSON message to MongoDB database.collection ‘tweets\_sentiment.social\_media\_sentiment’.



## Start Process Group GetNikeTweets



Once we start the process group, we can see data being read from the twitter API and consumed by kafka topic ‘tweets’. Now let’s see an example of a tweet being ingested into kafka.

1. Enter the process group ‘GetNikeTweets’ and right click on processor ‘Publishkafka.
2. Click on ‘View data provenance’
3. Click on the **i** icon for the event type ‘SEND’
4. Click on ‘CONTENT’ tab
5. On the Output Claim, choose VIEW

You will be able to see the data NiFi sent to the external process Kafka. The data below shows tweets dataset.



# Building Sentiment Analysis Model

For the purpose of this tutorial, Hortonworks provided with a set of tweets for building and training our sentiment classification model. We will create a Zeppelin notebook that uses Scala spark interpreter to clean our raw twitter data. Our Nifi flow already handles missing data. It drops the tweets that have empty message and publishes only complete tweet messages to kafka topic ‘tweet’. In this phase, we will classify our tweet sample data set into happy or sad sentiment based on the presence of these words.

## Prerequisites

Start HDFS, YARN, Zeppelin and Spark2 in Ambari

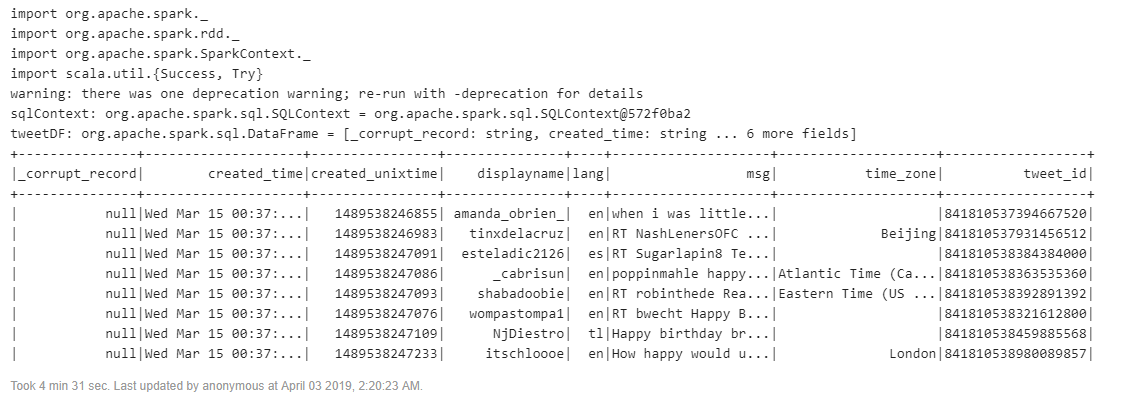
## Load data into Spark

import scala.util.{Success, Try}

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

var tweetDF = sqlContext.read.json("hdfs:///sandbox/tutorial-files/770/tweets\_staging/\*")

tweetDF.show()



In the following code we’ll clean-up the data to prevent bias in the model. We will use SparkSQL to retain equal number of happy and sad tweets and ignore the rest.

%spark2

var messages = tweetDF.select("msg")

println("Total messages: " + messages.count())

//Subset of messages that contain happy

var happyMessages = messages.filter(messages("msg").contains("happy"))

val countHappy = happyMessages.count()

println("Number of happy messages: " + countHappy)

//Subset of messages that contain sad

var unhappyMessages = messages.filter(messages("msg").contains(" sad"))

val countUnhappy = unhappyMessages.count()

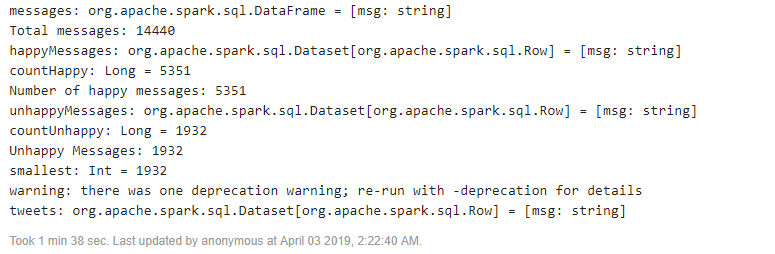
println("Unhappy Messages: " + countUnhappy)

//Lower boundary of the two subsets

val smallest = Math.min(countHappy, countUnhappy).toInt

//Create a dataset with equal parts happy and unhappy messages based on lower boundary

var tweets = happyMessages.limit(smallest).unionAll(unhappyMessages.limit(smallest))



First, we will label each tweet with 1 or 0 based on the presence of word ‘happy’ or ‘sad’ respectively. For convenience we convert the Spark Dataframe to an RDD which lets us easily transform data using the map function. An RDD is building block of spark. No matter which abstraction Dataframe or Dataset we use, internally final computation is done on RDDs. After labeling, we will remove these words from the messages and will use the rest of the collection of words to infer the sentiment.

%spark2

val messagesRDD = tweets.rdd

//We use scala's Try to filter out tweets that couldn't be parsed

val goodBadRecords = messagesRDD.map(

row =>{

Try{

val msg = row(0).toString.toLowerCase()

var isHappy:Int = 0

if(msg.contains(" sad")){

isHappy = 0

}else if(msg.contains("happy")){

isHappy = 1

}

var msgSanitized = msg.replaceAll("happy", "")

msgSanitized = msgSanitized.replaceAll("sad","")

//Return a tuple

(isHappy, msgSanitized.split(" ").toSeq)

}

}

)

//We use this syntax to filter out exceptions

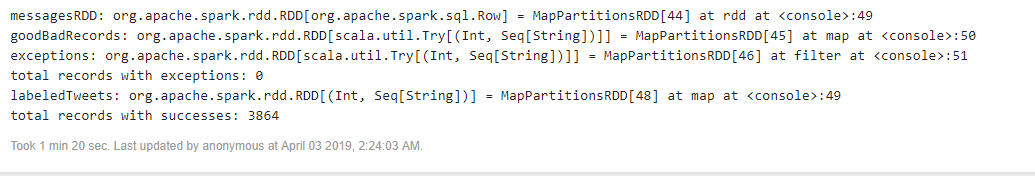
val exceptions = goodBadRecords.filter(\_.isFailure)

println("total records with exceptions: " + exceptions.count())

exceptions.take(10).foreach(x => println(x.failed))

var labeledTweets = goodBadRecords.filter((\_.isSuccess)).map(\_.get)

println("total records with successes: " + labeledTweets.count())

Most of ML algorithms require us to represent input data as real number matrix. Process of converting raw data to real number matrix is called feature engineering, and hashing trick is a feature engineering technique. In Spark we’re using HashingTF for feature hashing.

Gradient Boosting expects as input a vector (feature array) of fixed length, so we need a way to convert our tweets into some numeric vector that represents that tweet. A standard way to do this is to use the hashing trick, in which we hash each word and index it into a fixed-length array. What we get back is an array that represents the count of each word in the tweet. This approach is called the bag of words model, which means we are representing each sentence or document as a collection of discrete words and ignore grammar or the order in which words appear in a sentence.

%spark2

val hashingTF = new HashingTF(2000)

//Map the input strings to a tuple of labeled point + input text

val input\_labeled = (labeledTweets.map(

t => (t.\_1, hashingTF.transform(t.\_2)))

.map(x => new LabeledPoint((x.\_1).toDouble, x.\_2)))

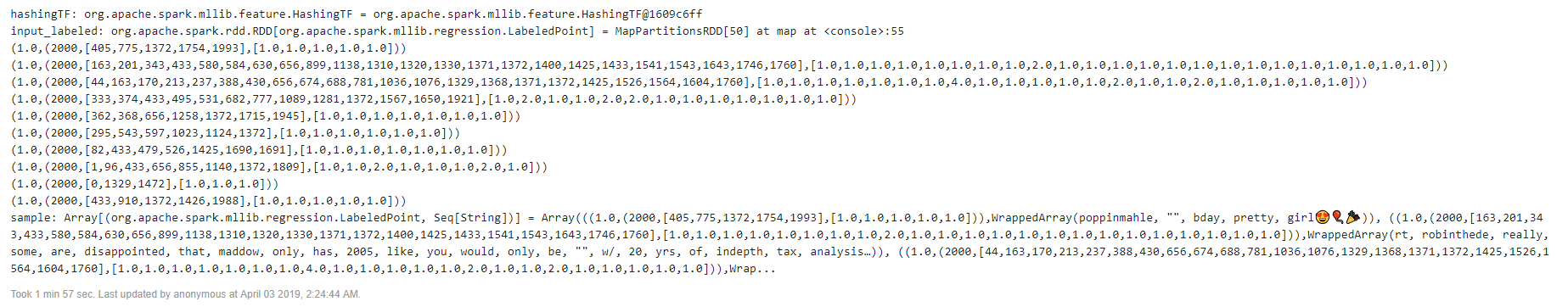
input\_labeled.take(10).foreach(println)

//We're keeping the raw text for inspection later

var sample = (labeledTweets.take(1000).map(

t => (t.\_1, hashingTF.transform(t.\_2), t.\_2))

.map(x => (new LabeledPoint((x.\_1).toDouble, x.\_2), x.\_3)))



## Gradient Boosting Classification Model

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. (Wikipedia definition). Boosting is a method of converting weak learners into strong learners. The tuning parameters we’re using here are:

-number of iterations (passes over the data)

-Max Depth of each decision tree

In practice when building machine learning models we usually have to test different settings and combinations of tuning parameters until we find the model that fits the data best. For this reason, it’s usually best to first train the model on a subset of data or with a small number of iterations. This lets us quickly experiment with different tuning parameter combinations.

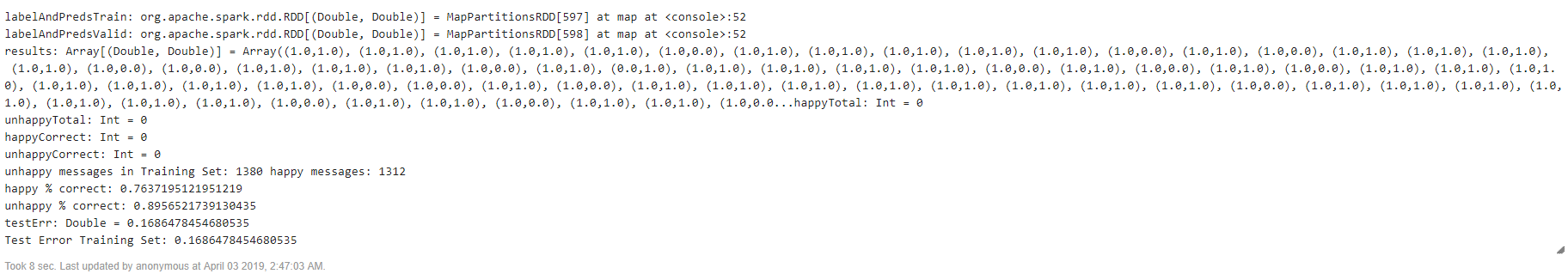
1. // Split the data into training and validation sets (30% held out for validation testing)
2. val splits = input\_labeled.randomSplit(Array(0.7, 0.3))
3. val (trainingData, validationData) = (splits(0), splits(1))
4. val boostingStrategy = BoostingStrategy.defaultParams("Classification")
5. boostingStrategy.setNumIterations(20) //number of passes over our training data
6. boostingStrategy.treeStrategy.setNumClasses(2) //We have two output classes: happy and sad
7. boostingStrategy.treeStrategy.setMaxDepth(5)
8. //Depth of each tree. Higher numbers mean more parameters, which can cause overfitting.
9. //Lower numbers create a simpler model, which can be more accurate.
10. val model = GradientBoostedTrees.train(trainingData, boostingStrategy)

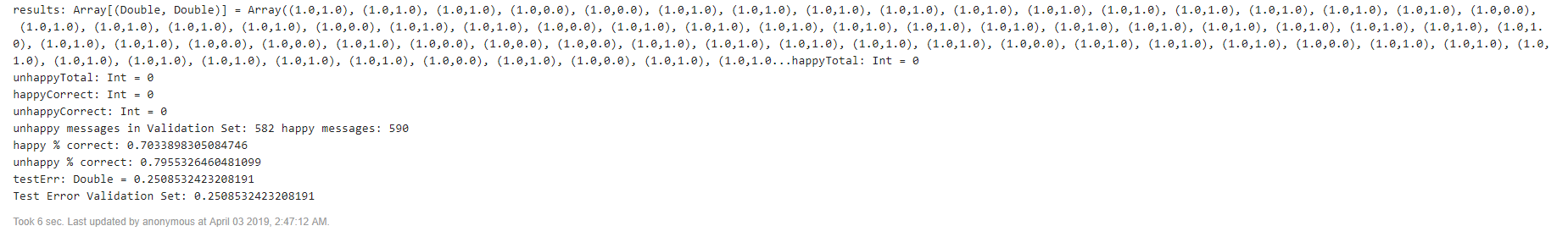
Let’s evaluate the model to see how it performed against our training and test set.

1. %spark2
2. // Evaluate model on test instances and compute test error
3. var labelAndPredsTrain = trainingData.map { point =>
4. val prediction = model.predict(point.features)
5. Tuple2(point.label, prediction)
6. }
8. var labelAndPredsValid = validationData.map { point =>
9. val prediction = model.predict(point.features)
10. Tuple2(point.label, prediction)
11. }
13. //Since Spark has done the heavy lifting already, lets pull the results back to the driver machine.
14. //Calling collect() will bring the results to a single machine (the driver) and will convert it to a Scala array.
16. //Start with the Training Set
17. val results = labelAndPredsTrain.collect()
19. var happyTotal = 0
20. var unhappyTotal = 0
21. var happyCorrect = 0
22. var unhappyCorrect = 0
23. results.foreach(
24. r => {
25. if (r.\_1 == 1) {
26. happyTotal += 1
27. } else if (r.\_1 == 0) {
28. unhappyTotal += 1
29. }
30. if (r.\_1 == 1 && r.\_2 ==1) {
31. happyCorrect += 1
32. } else if (r.\_1 == 0 && r.\_2 == 0) {
33. unhappyCorrect += 1
34. }
35. }
36. )
37. println("unhappy messages in Training Set: " + unhappyTotal + " happy messages: " + happyTotal)
38. println("happy % correct: " + happyCorrect.toDouble/happyTotal)
39. println("unhappy % correct: " + unhappyCorrect.toDouble/unhappyTotal)
41. val testErr = labelAndPredsTrain.filter(r => r.\_1 != r.\_2).count.toDouble / trainingData.count()
42. println("Test Error Training Set: " + testErr)

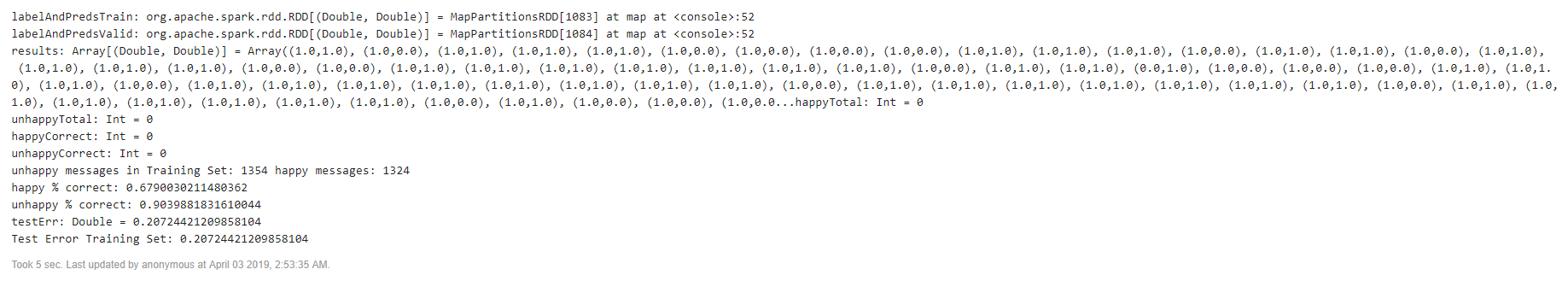
45. //Compute error for validation Set
46. val results = labelAndPredsValid.collect()
48. var happyTotal = 0
49. var unhappyTotal = 0
50. var happyCorrect = 0
51. var unhappyCorrect = 0
52. results.foreach(
53. r => {
54. if (r.\_1 == 1) {
55. happyTotal += 1
56. } else if (r.\_1 == 0) {
57. unhappyTotal += 1
58. }
59. if (r.\_1 == 1 && r.\_2 ==1) {
60. happyCorrect += 1
61. } else if (r.\_1 == 0 && r.\_2 == 0) {
62. unhappyCorrect += 1
63. }
64. }
65. )
66. println("unhappy messages in Validation Set: " + unhappyTotal + " happy messages: " + happyTotal)
67. println("happy % correct: " + happyCorrect.toDouble/happyTotal)
68. println("unhappy % correct: " + unhappyCorrect.toDouble/unhappyTotal)
70. val testErr = labelAndPredsValid.filter(r => r.\_1 != r.\_2).count.toDouble / validationData.count()
71. println("Test Error Validation Set: " + testErr)

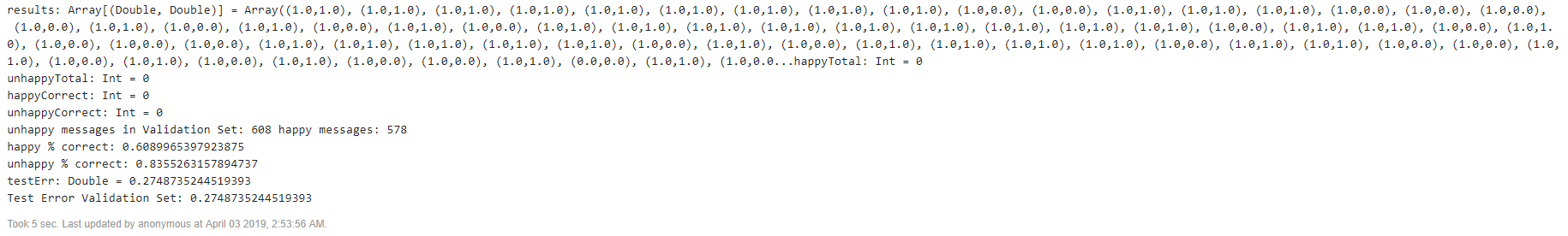
 Depth: 5



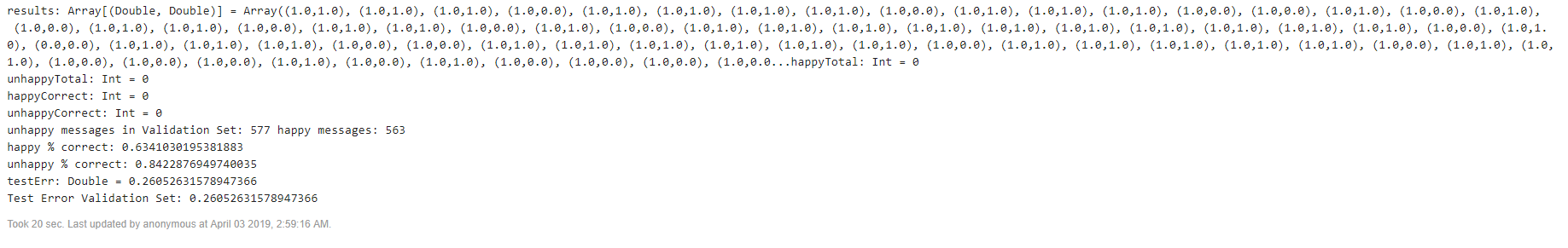
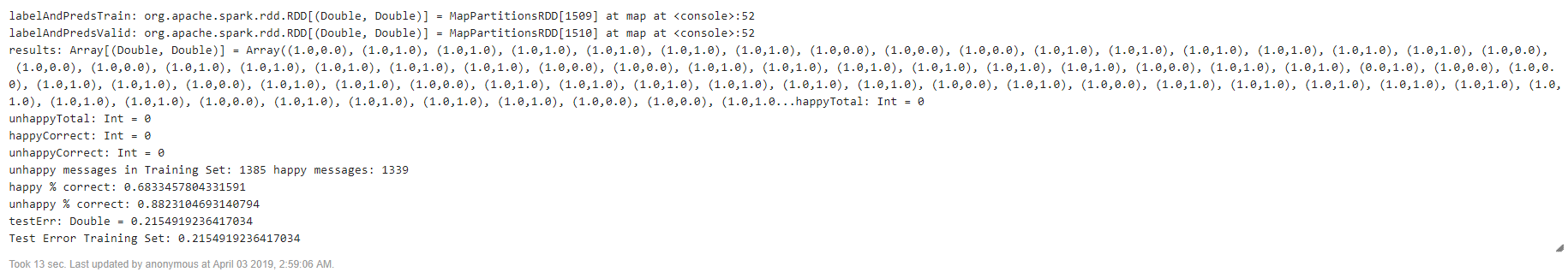


Depth:4

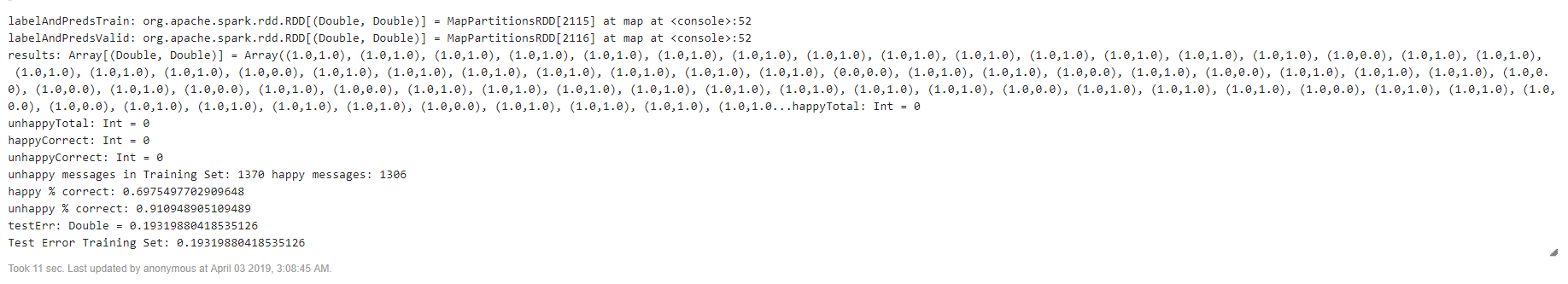


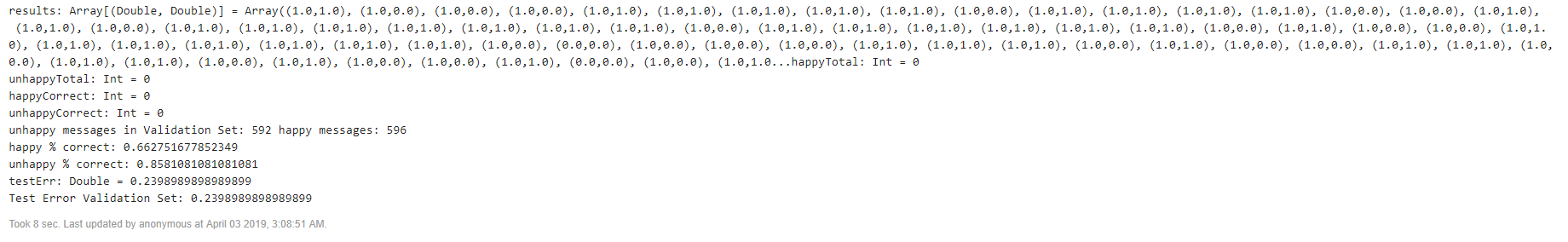


Depth:3

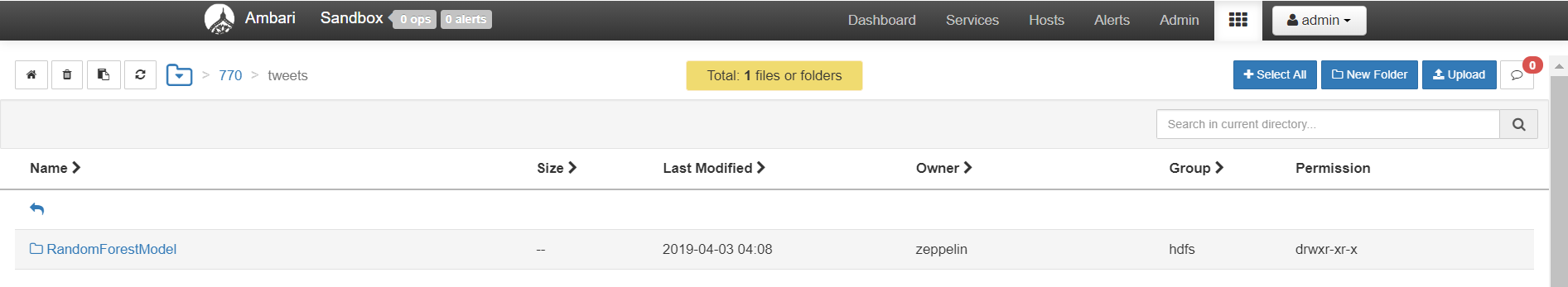


Depth:4 Iterations: 25

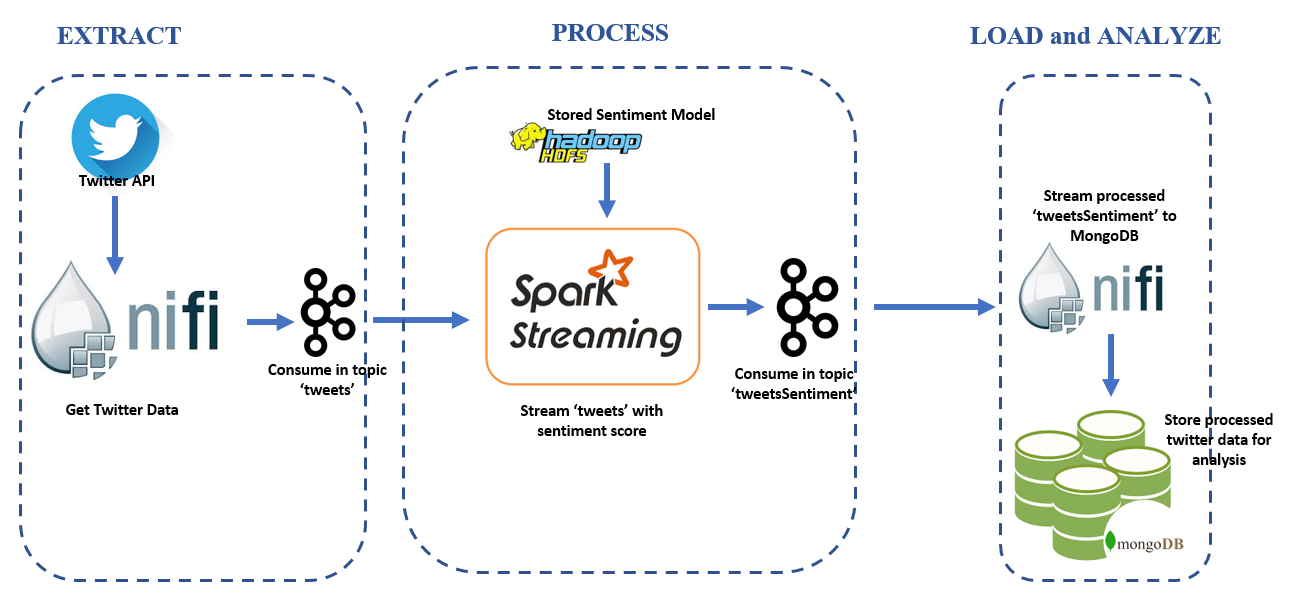


Once our model is as accurate as we can make it, we can export it for production use. Models trained with Spark can be easily loaded back into a Spark Structured Streaming workflow for use in production.

1. %spark2
2. model.save(sc, "hdfs:///sandbox/tutorial-files/770/tweets/RandomForestModel")



# Spark Streaming Application to Deploy Model



We configured our Spark streaming application as follows:

spark {

kafkaBrokers {

kafkaBrokerHDF: "sandbox-hdf.hortonworks.com:6667"

kafkaBrokerHDP: "sandbox-hdp.hortonworks.com:6667"

}

appName = "SentimentModel"

messageFrequency = 200 //milliseconds

modelLocation = "hdfs:///sandbox/tutorial-files/770/tweets/RandomForestModel"

kafkaTopics {

tweetsRaw: "tweets"

tweetsWithSentiment: "tweetsSentiment"

}

}

Our Spark Streaming Application contains two classes. ‘Predictor’ class converts the tweet into a vector using Hashing transformation and predicts the sentiment. The Collect class collects the tweet, applies the predict object from Predictor class and appends the sentiment score to the outgoing tweet.

**Predictor.scala**

1. class Predictor(model: GradientBoostedTreesModel){

// Function to predict the sentiment on vectorized tweet msg

1. def predict(tweet:String): Double ={
2. if(tweet == null || tweet.length == 0)
3. throw new RuntimeException("Tweet is null")
4. val features = vectorize(tweet)
5. return model.predict(features)
6. }

// Function to vectorize raw tweet msg

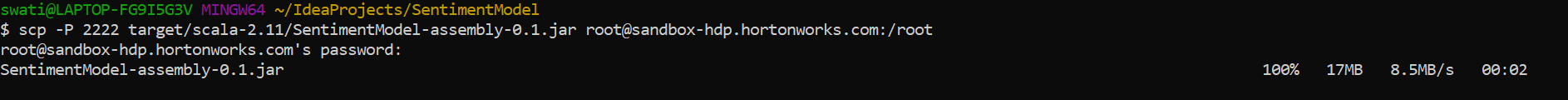
1. val hashingTF = new HashingTF(2000)
2. def vectorize(tweet:String):Vector={
3. hashingTF.transform(tweet.split(" ").toSeq)
4. }
6. }

***Collect.scala***

1. val options = new CollectOptions(
2. config.getString("spark.kafkaBrokers.kafkaBrokerHDF"),
3. config.getString("spark.kafkaBrokers.kafkaBrokerHDP"),
4. config.getString("spark.kafkaTopics.tweetsRaw"),
5. config.getString("spark.kafkaTopics.tweetsWithSentiment"),
6. config.getString("spark.appName"),
7. config.getString("spark.modelLocation")
8. )
10. val spark = SparkSession
11. .builder
12. .appName("DeploySentimentModel")
13. .getOrCreate()
14. spark.sparkContext.setLogLevel("ERROR")// Create DataSet representing the stream of input lines from kafka
16. val rawTweets = spark
17. .readStream
18. .format("kafka")
19. .option("kafka.bootstrap.servers", options.kafkaBrokerHDP)
20. .option("subscribe", options.tweetsTopic)
21. .load()
22. .selectExpr("CAST(value AS STRING)")
23. .as[String]
24. // Create DataSet representing the stream of input lines from kafka
25. val rawTweets = spark
26. .readStream
27. .format("kafka")
28. .option("kafka.bootstrap.servers", options.kafkaBrokerHDP)
29. .option("subscribe", options.tweetsTopic)
30. .load()
31. .selectExpr("CAST(value AS STRING)")
32. .as[String]
34. rawTweets.printSchema()
35. //Our Predictor class can't be serialized, so we're using mapPartition to create a new model instance for each partition.
36. val tweetsWithSentiment = rawTweets.mapPartitions((iter) => {
37. val pred = new Predictor(model
38. val parser = new JsonParser()
39. iter.map(
40. tweet =>
41. //For error handling, we're mapping to a Scala Try and filtering out records with errors.
42. Try {
43. val element = parser.parse(tweet).getAsJsonObject
44. val msg = element.get("text").getAsString
45. val sentiment = pred.predict(msg)
46. element.addProperty("sentiment", pred.predict(tweet))
47. val json = element.toString
48. println(json)
49. json
50. }
51. ).filter(\_.isSuccess).map(\_.get)
52. })
54. val query = tweetsWithSentiment.writeStream
55. .outputMode("append")
56. .format("console")
57. .start()
59. //Push back to Kafka
60. val kafkaProps = new Properties()
61. //props.put("metadata.broker.list", options.kafkaBrokerList)
62. kafkaProps.put("bootstrap.servers", options.kafkaBrokerHDF)
63. kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
64. kafkaProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")
66. tweetsWithSentiment
67. .writeStream
68. .foreach(
69. new ForeachWriter[(String)] {
71. //KafkaProducer can't be serialized, so we're creating it locally for each partition.
72. var producer:KafkaProducer[String, String] = null
74. override def process(value: (String)) = {
75. val message = new ProducerRecord[String, String](options.tweetsWithSentimentTopic, null,value)
76. println("sending windowed message: " + value)
77. producer.send(message)
78. }
80. override def close(errorOrNull: Throwable) = ()
82. override def open(partitionId: Long, version: Long) = {
83. producer = new KafkaProducer[String, String](kafkaProps)
84. true
85. }
86. }).start()
88. query.awaitTermination()

**Copying Sentiment model to HDFS**

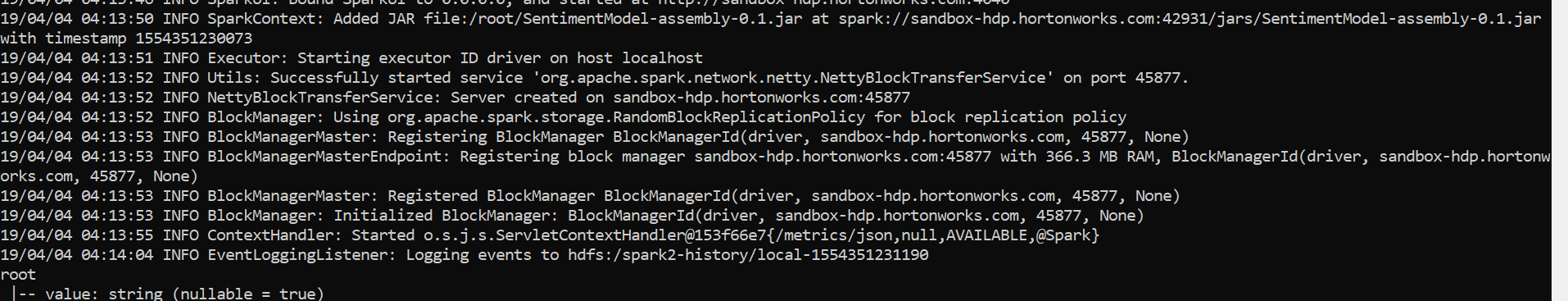
We copied the jar version of our spark streaming application into HDFS



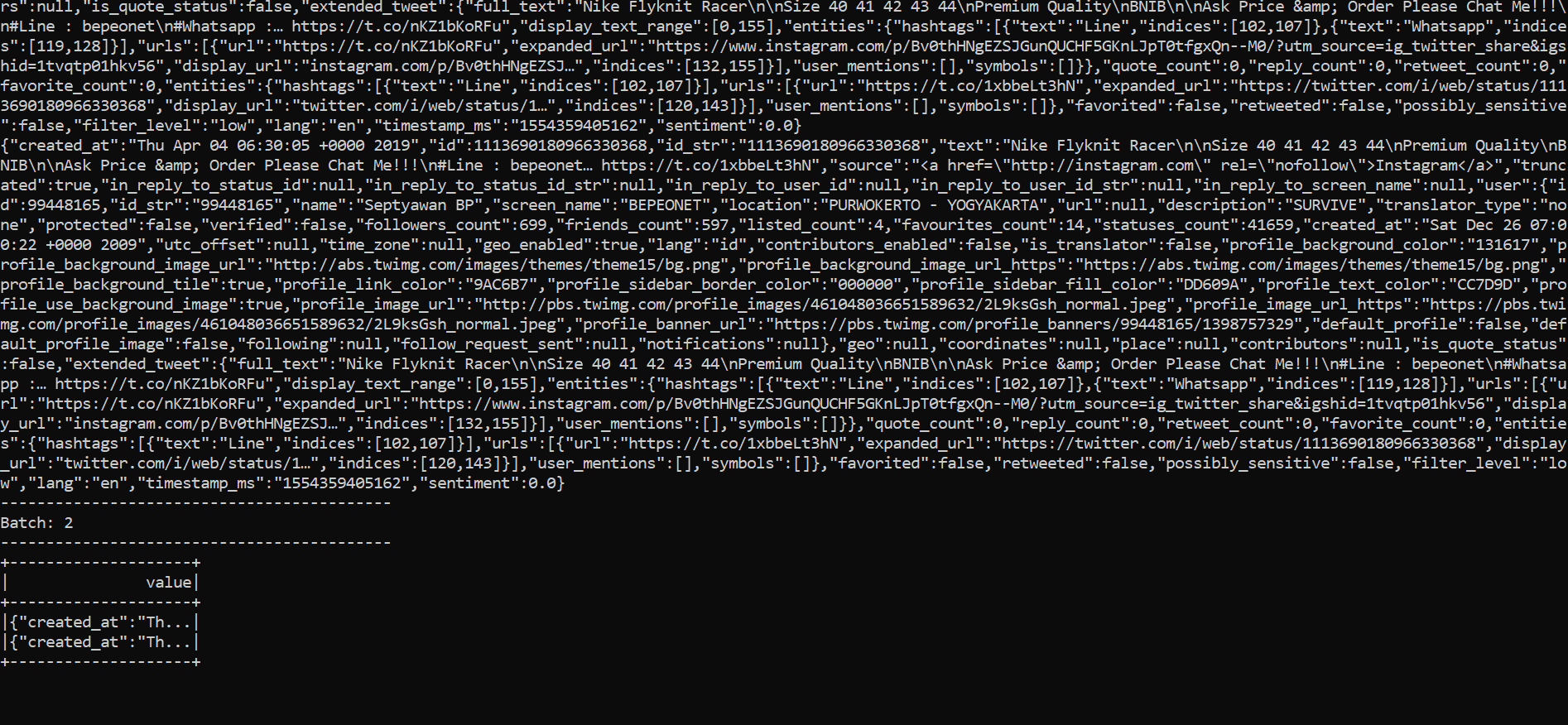
Now we will deploy the jar file on our local HDP and can see the output as Spark scores each tweet.

1. /usr/hdp/current/spark2-client/bin/spark-submit --class "main.scala.Collect" --master local[4] root/SentimentModel-assembly-0.1.jar

 Once the jar is deployed, the application starts listening for tweets

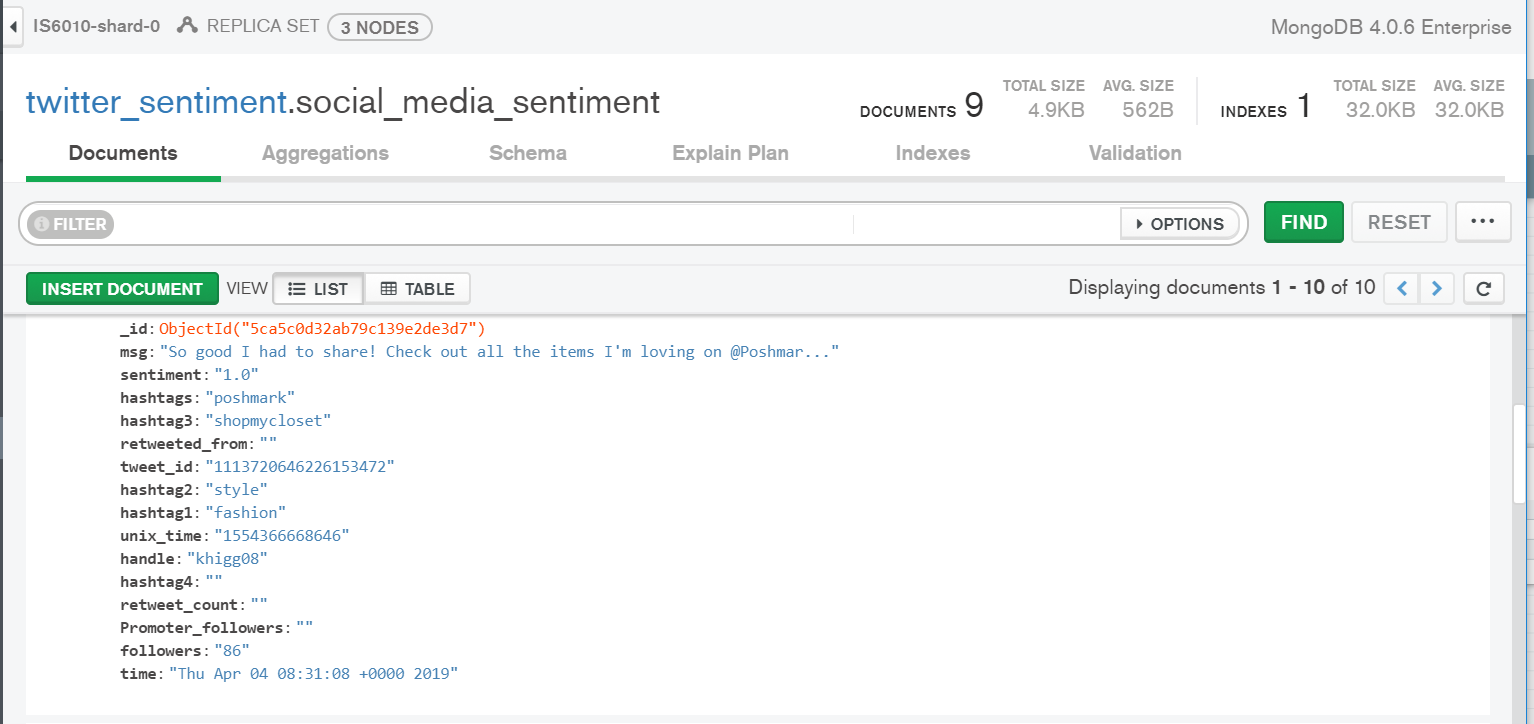


We can see below that sentiment score is being



# Data Analysis in MongoDB

When we load data into MongoDB, we see that all attributes have default datatype as String. This is because Nifi Processor AttributestoJSON converts all attributes to String. We can use the aggregation framework in MongoDB to further process data into our desired format. Below is one of the first records that was streamed through our flow and loaded into MongoDB.



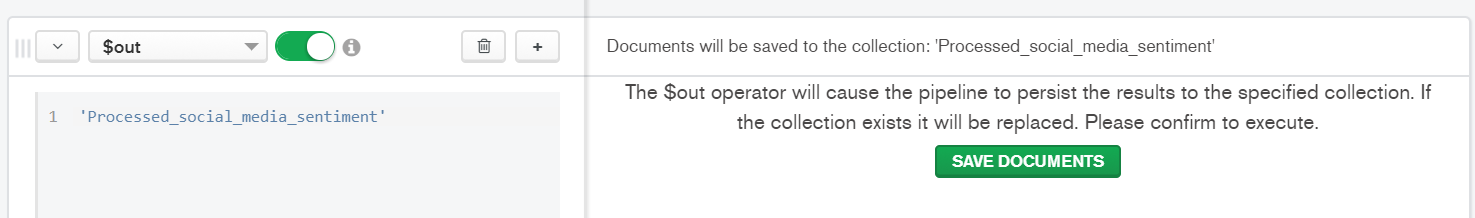
We will now transform this document and all the other documents into desired output. We will use the aggregation framework and create a data pipeline as follows



Query:

1. $project:/\*\*
2. \* specifications - The fields to
3. \* include or exclude.
4. \*/
5. {
6. tweet\_id:'$tweet\_id',
7. created\_at:{$dateFromString:{dateString:'$time'}},
8. user:'$handle',
9. followers:{$toInt:'$followers'},
10. msg:'$msg',
11. hashtags:{$concat:['$hashtags',',','$hashtag1',',','$hashtag2',',','$hashtag3',',','$hashtag4']},
12. sentiment:{$toInt:{$toDecimal:'$sentiment'}},
13. retweeted\_from:'$retweeted\_from',
14. Promoter\_followers:{
15. $cond: { if: {$gt:[{$strLenCP:'$Promoter\_followers'},0]}, then: {$toInt:'$Promoter\_followers'}, else: 0 }
16. },
17. retweet\_count:{
18. $cond: { if: {$gt:[{$strLenCP:'$retweet\_count'},0]}, then: {$toInt:'$retweet\_count'}, else: 0 }
20. }
21. }
23. $out:’Processed\_social\_media\_sentiment’

 We will store the result in a separate collection.



This is how the processed document looks like

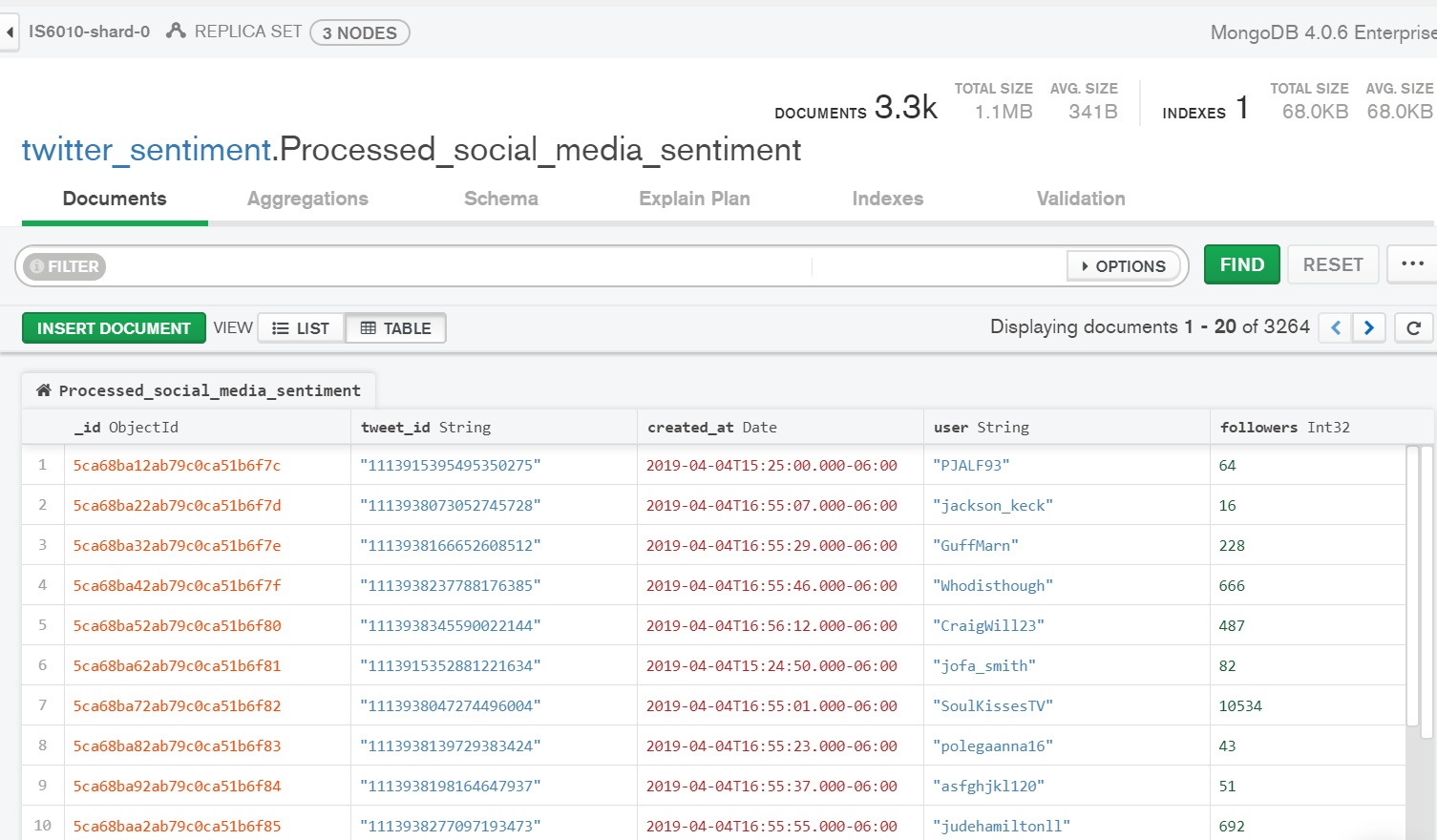


Upon quickly going through some of the documents, I realized that the retweeted\_from , retweet\_count and promoter\_followers fields were empty. This is because we filtered tweets with empty messages in the Nifi flow. However, this is an important attribute as we can gauge the reach of a tweets based on its retweet count. So let’s modify the nifi flow to include retweets as well.

We modified the condition from Dataflow1 in RouteonAttribute for **${twitter.handle:isEmpty():not()}**



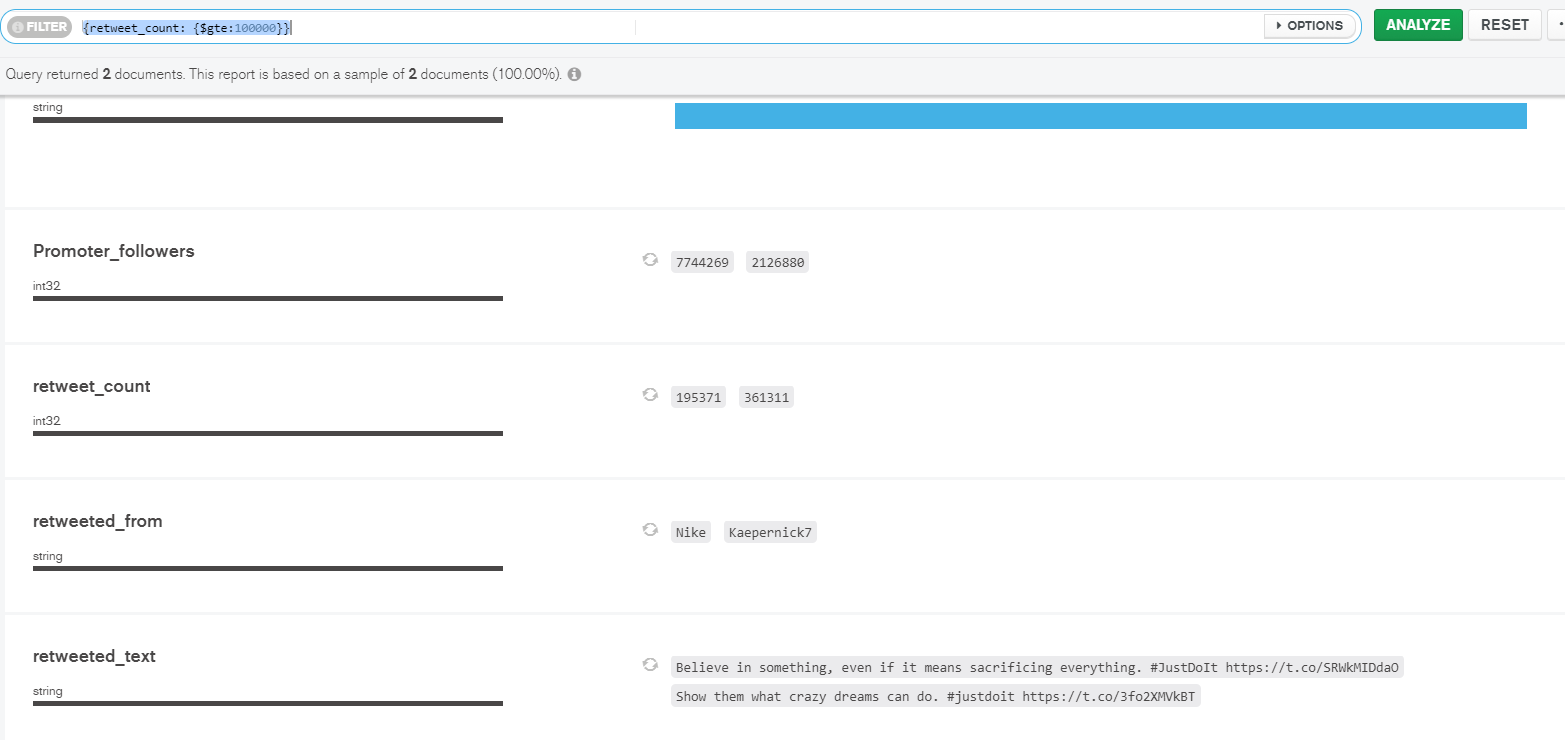
Let’s rerun the flow to analyze new tweet documents. We can see that we are getting are now recording retweets as well. After running for few minutes we have over 3.3K records or documents.



**Some Insights:**

**Most popular retweets:**

{retweet\_count: {$gte:100000}}



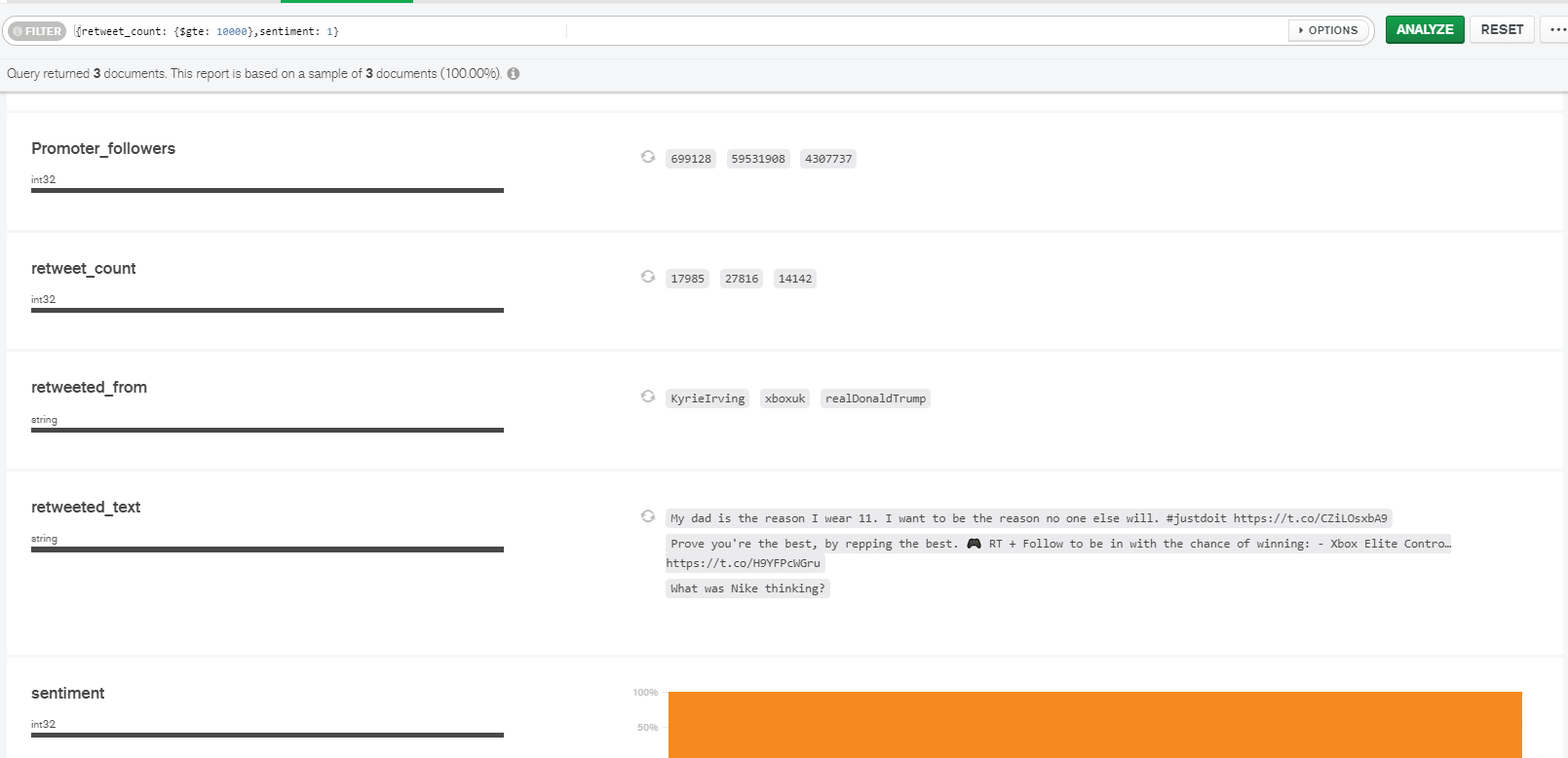
Both their most popular ads were received negatively. First is the Ad with Kaepernick. Although Nike made [huge profits](https://www.vox.com/2018/9/24/17895704/nike-colin-kaepernick-boycott-6-billion) after releasing this ad, it is causing a huge debate around whether kneeling during the national anthem was insulting for the American soldiers or protest again police brutality. However, this is what Nike is famous for. They win the odds by striking a debate about important topics.

Their other ad focusing on women athletes also has been received well with the audience and has been retweeted 195K times since its release in Feb 2019.

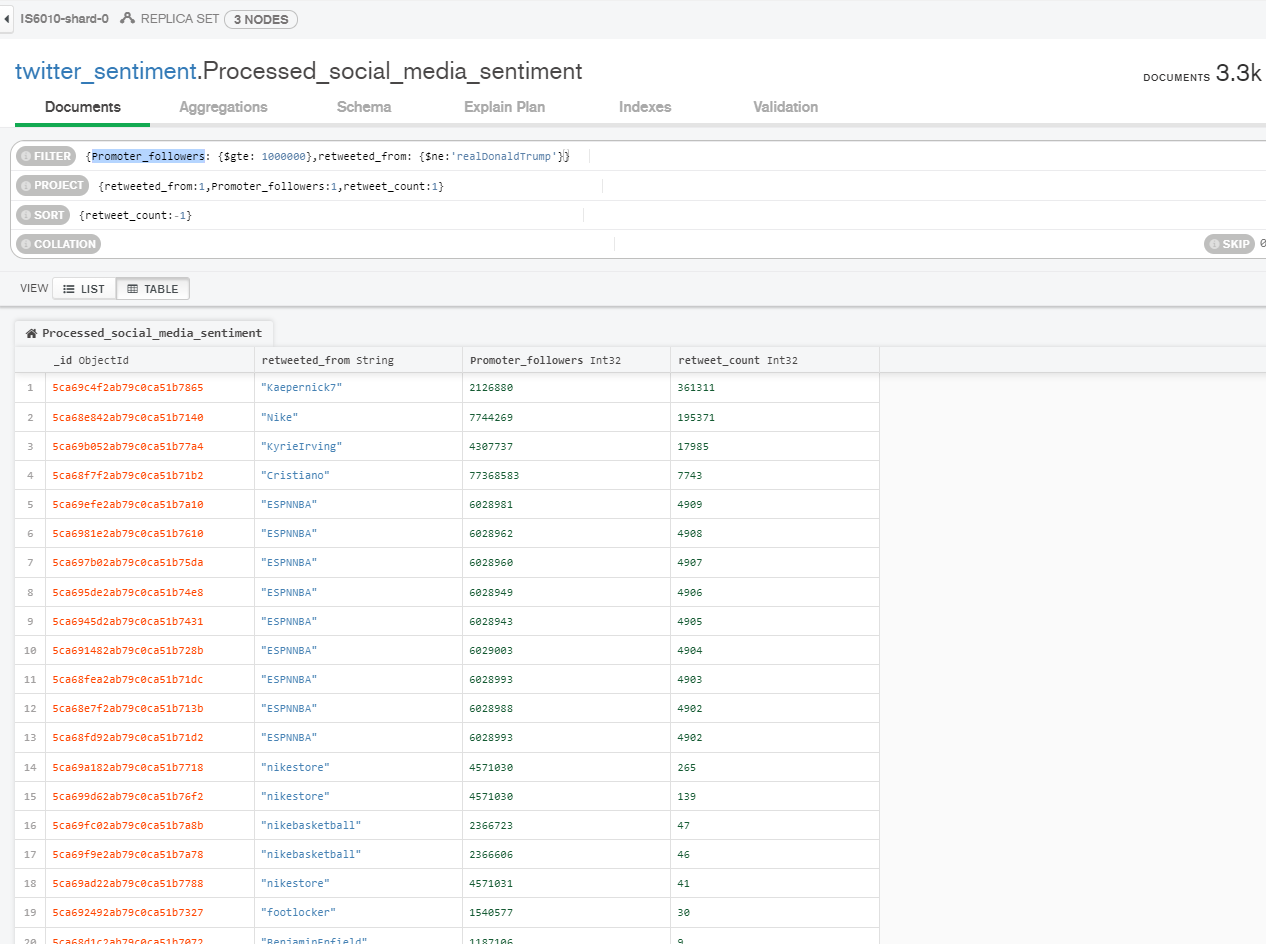
It is also important to understand that these ads with negative sentiment have been hugely popular. We can safely say that ads that bring out bias have been beneficial for Nike in striking some kind of chord with the audience.

Since our sentiment is only 1 and 0, it also appears that tweets with neutral sentiment are also getting recorded as 0. So lets run some analysis on tweets with sentiment score 1.

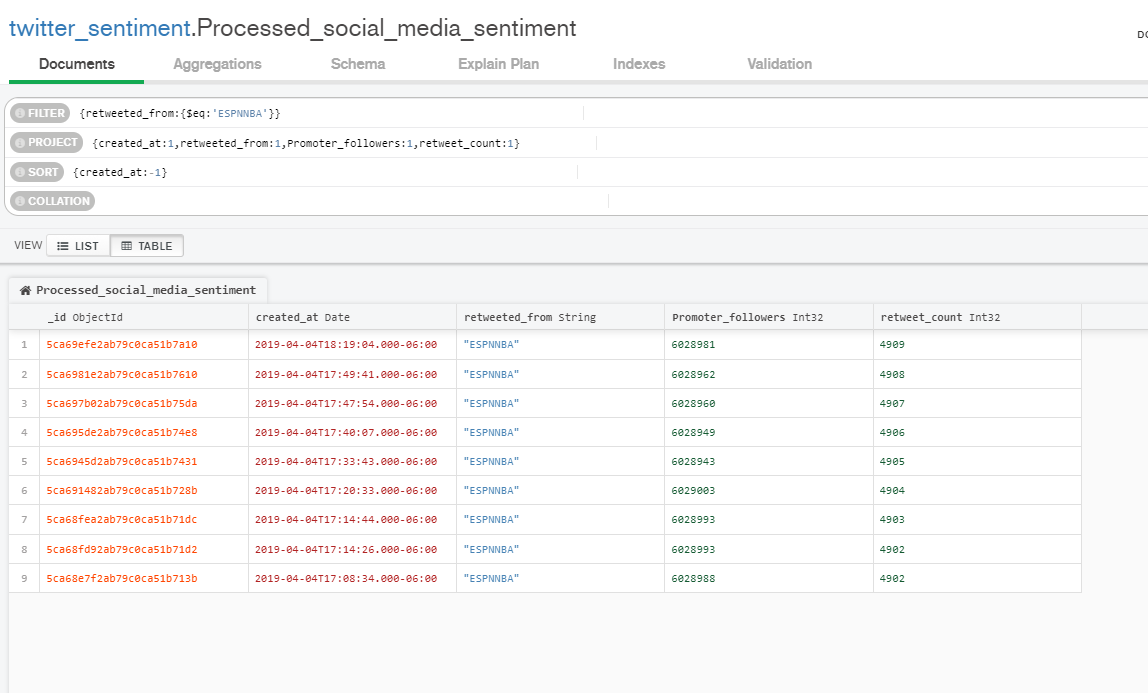
**Most popular positive tweet**



The Nike Ad by Kyrie Irving was retweeted 17K times. It is still popular since its release in November 2018.



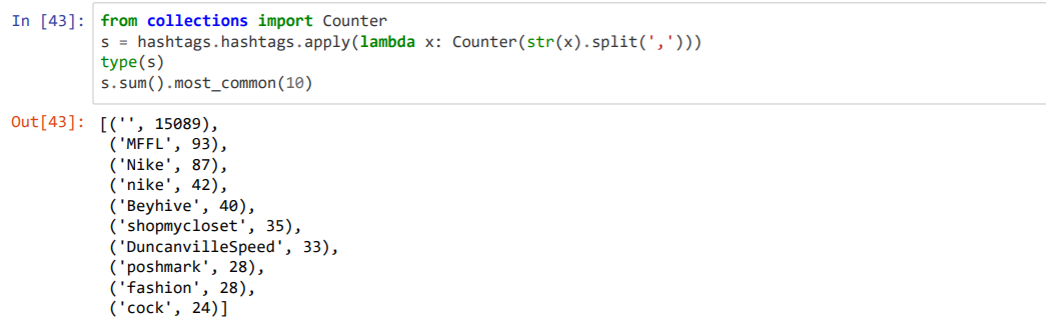
Also we can see the most popular influencer handles based on the analysis of only 3.3k documents on twitter are Kaepernick, Nike, KyrieIrving, Cristiano, ESPNNBA, nikestore, nikebasketball and footlocker.



Here we can see that ESPNNBA has had a decrease of followers in a few minutes from 6028988 to 6028981. We also see a fluctuation in the number of followers over time.The number is small due to a small time window of the data captured.

Most gathering the count of most popular hashtags, I exported the hashtags column from Processed\_social\_media\_sentiment and loaded into python for analysis





MFFL is a news handle for Dallas Maverick. Nike is the apparel and shoes partner of this popular basketball team.

# Resources

What Is Kafka? <https://bernardmarr.com/default.asp?contentID=1525>

MongoDB Aggregation Framework Course: <https://university.mongodb.com/mercury/M121/2019_March/overview>

Aggregation Pipeline Operators: <https://docs.mongodb.com/manual/reference/operator/aggregation/>

Building sentiment analysis application: <https://hortonworks.com/tutorial/building-a-sentiment-analysis-application/>